

# Outstanding User Interfaces with Shiny

David Granjon

2020-05-13



# Contents

<b>Prerequisites</b>	<b>7</b>
Disclaimer . . . . .	7
Is this book for me? . . . . .	7
Related content . . . . .	7
<b>1 Introduction</b>	<b>9</b>
<b>Survival Kit</b>	<b>13</b>
<b>2 HTML</b>	<b>15</b>
2.1 HTML Basics . . . . .	15
2.2 Tag attributes . . . . .	16
2.3 HTML page: skeleton . . . . .	17
2.4 About the DOM . . . . .	18
2.5 Preliminary introduction to CSS and JavaScript . . . . .	18
<b>3 JavaScript</b>	<b>23</b>
3.1 Introduction . . . . .	23
3.2 Setup . . . . .	25
3.3 Programming with JS: basis . . . . .	27
3.4 jQuery . . . . .	35
<b>4 Shiny, under the hood</b>	<b>41</b>
4.1 Shiny dependencies . . . . .	41
4.2 Shiny hidden gems . . . . .	42

<b>htmltools</b>	<b>55</b>
<b>5 htmltools overview</b>	<b>57</b>
5.1 HTML Tags . . . . .	57
5.2 Notations . . . . .	57
5.3 Adding new tags . . . . .	57
5.4 Alternative way to write tags . . . . .	58
5.5 Playing with tags . . . . .	58
<b>6 Dependency utilities</b>	<b>65</b>
6.1 The dirty approach . . . . .	65
6.2 The clean approach . . . . .	67
6.3 Another example: Importing HTML dependencies from other packages . . . . .	68
6.4 Suppress dependencies . . . . .	71
<b>7 Other tools</b>	<b>73</b>
7.1 CSS . . . . .	73
<b>Practice</b>	<b>77</b>
<b>8 Template selection</b>	<b>79</b>
<b>9 Define dependencies</b>	<b>81</b>
9.1 Discover the project . . . . .	81
9.2 Identify mandatory dependencies . . . . .	83
9.3 Bundle dependencies . . . . .	84
<b>10 Template skeleton</b>	<b>87</b>
10.1 Identify template elements . . . . .	87
10.2 Design the page layout . . . . .	87
<b>11 Develop custom input widgets</b>	<b>105</b>
11.1 How does Shiny handle inputs? . . . . .	105
11.2 How to add new input to Shiny? . . . . .	105

<i>CONTENTS</i>	5
<b>12 Testing templates elements</b>	<b>107</b>



# Prerequisites

- Be familiar with Shiny
- Basic knowledge in HTML and JavaScript is a plus but not mandatory

## Disclaimer

This book is not an HTML/Javascript/CSS course! Instead, it provides a *survival kit* to be able to customize Shiny. I am sure however that readers will want to explore more about these topics.

## Is this book for me?

You should read this book if you answer yes to the following questions:

- Do you want to know how to develop outstanding shiny apps?
- Have you ever wondered how to develop new input widgets?

## Related content

See the RStudio Cloud dedicated project.

```
library(shiny)
library(shinydashboard)
library(shiny.semantic)
library(cascadess)
library(htmltools)
library(purrr)
library(magrittr)
```





# Chapter 1

## Introduction

There are various Shiny focused resources introducing basic as well as advanced topics such as modules and Javascript/R interactions, however, handling advanced user interfaces design was never an emphasis. Clients often desire custom templates, yet this generally exceeds core features of Shiny (not out of the box).

Generally, R App developers lack a significant background in web development and often find this requirement overwhelming. It was this sentiment that motivated writing this book, namely to provide readers the necessary knowledge to extend Shiny's layout, input widgets and output elements. This project officially started at the end of 2018 but was stopped when Joe Cheng revealed the upcoming Mastering Shiny Book. Fortunately, the later, does not cover a lot regarding how to customize Shiny user interfaces and our book will deal with the reactive programming concepts, nor user feedbacks or modules. Besides, this book may constitute a good complement to the work in progress Engineering Production-Grade Shiny Apps by the ThinkR team, where the link between Shiny and CSS/JavaScript is covered.

This book is organized into four parts.

- We first go through the basics of HTML, JavaScript and jQuery and finish with a chapter dedicated to the partially hidden features of Shiny, yet so fun
- In part 2, we dive into the `{htmltools}` package, providing functions to create and manipulate shiny tags as well as manage dependencies
- Part 3 focuses on the development of a new template for Shiny by demonstrating examples from the `{tablerDash}`, `{bs4Dash}` and `{shinyMobile}` packages. These, and more may be explored further as part of the Rinterface project.
- Part 4 present some tools of the R community, like `{fresh}`, to beautify apps with only few lines of code



# Survival Kit



This part will give you basis in HTML, JavaScript to get started...



## Chapter 2

# HTML

This chapter provides a short introduction to the HTML language. As a quick example, open up RStudio and perform the following:

- Load shiny with `library(shiny)`
- Execute `p("Hello World")`

Notice the output format is an example of an HTML tag!

### 2.1 HTML Basics

HTML (Hypertext Markup Language) is derived from SGML (Standard Generalized markup Language). An HTML file contains tags that may be divided into 2 categories:

- paired-tags: the text is inserted between the opening and the closing tag
- closing-tags

```
<!-- paired-tags -->
<p></p>
<div></div>

<!-- self-closing tags -->
<iframe/>
<img/>
<input/>
<br/>
```

Tags may be divided into 3 categories, based on their role:

- structure tags: they constitute the skeleton of the HTML page (`<title></title>`, `<head></head>`, `<body></body>`)
- control tags: script, inputs and buttons (and more). Their role is to include external resources, provide interactivity with the user
- formatting tags: to control the size, font of the wrapped text

Finally, we distinguish block and inline elements:

- block elements may contain other tags and take the full width (block or inline). `<div></div>` is the most commonly used block element. All elements of a block are printed on top of each others
- inline elements (for instance `<span></span>`, `<a></a>`) are printed on the same line. They can not contain block tags but may contain other nested inline tags. In practice, we often see `<a><span></span></a>`
- inline-block elements allow to insert block element in an inline

Consider the following example. This is clearly a bad use of HTML conventions since an inline tag can not host block elements.

```
<span>
  <div><p>Hello World</p></div>
  <div></div>
</span>
```

Importantly, `<div>` and `<span>` don't have any semantic meaning, contrary to `<header>` and `<footer>`, which allow to structure the HTML page.

## 2.2 Tag attributes

Attributes are text elements allowing to specify some properties of the tag. For instance for a link tag (`<a></a>`), we actually expect more than just the tag itself: a target url and how to open the new page ... In all previous examples, tags don't have any attributes. Yet, there exist a large range of attributes and we will only see 2 of them for now (the reason is that these are the most commonly used in CSS and JavaScript):

- class: may be shared between multiple tags
- id: each must be unique

```
<div class="awesome-item" id="myitem"></div>
<!-- the class awesome-item may be applied to multiple tags -->
<span class="awesome-item"></span>
```



Both attributes are widely used by CSS and JavaScript (see Chapter 3 with the jQuery selectors) to apply a custom style to a web page. Class attributes apply to multiple elements, however the id attribute is restricted to only one item.

Interestingly, there exists another attribute category, known as non-standard attributes like `data-toggle`. We will see them later in the book (see Chapter 10).

## 2.3 HTML page: skeleton

An HTML page is a collection of tags which will be interpreted by the web browser step by step. The simplest HTML page may be defined as follows:

```
<!DOCTYPE HTML>
<html>
  <head>
    <!-- head content here -->
  </head>
  <body>
    <!-- body content here -->
  </body>
</html>
```

- `<html>` is the main wrapper
- `<head>` and `<body>` are the 2 main children

`<head>` contains dependencies like styles and JavaScript files (but not only), `<body>` contains the page content. We will see later that JavaScript files are often added just before the end of the `<body>`.

Only the body content is displayed on the screen!

Let's write the famous Hello World in html:

```
<!DOCTYPE HTML>
<html>
  <head>
    <!-- head content here -->
  </head>
  <body>
    <p>Hello World</p>
  </body>
</html>
```

In order to preview this page in a web browser, you need to save the above snippet to a script `hello-world.html` and double-click on it. It will open with your default web browser.

## 2.4 About the DOM

The DOM stands for “Document Object Model”, is a convenient representation of the html document. There actually exists multiple DOM types, namely DOM-XML and DOM-HTML but we will only focus on the later (in the following DOM is DOM-HTML). If we consider the last example (Hello World), the associated DOM tree may be inspected in Figure 2.1.

### 2.4.1 Visualizing the DOM: the HTML inspector

Below, we introduce a tool that we are going to be a valuable ally during our ambitious quest to beautiful shiny user interfaces. In this chapter, we restrict the description to the first panel of the HTML inspector <sup>1</sup>. This feature is available in all web browser but we will only focus on Chrome.

- Open the hello-world.html example in a web browser (google chrome here)
- Right-click to open the HTML inspector (developer tools must be enabled if it is not the case)

The HTML inspector is a convenient tool to explore the structure of the current HTML page. On the left-hand side, the DOM tree is displayed and we clearly see that `<html>` is the parent of `<head>` and `<body>`. `<body>` has also 1 child, that is `<p>`. We didn’t mention this yet but we can preview any style (CSS) associated to the selected element on the right panel as well as Event Listeners (JavaScript). We will discuss that in the next chapter.

## 2.5 Preliminary introduction to CSS and JavaScript

CSS and JavaScript are tools to enhance an HTML page.

### 2.5.1 HTML and CSS

CSS (Cascading Style Sheets) changes the style of HTML tags by targeting specific classes or ids. For instance, if we want all p tags to have red color we will use:

---

<sup>1</sup>As shown in Figure 2.1, the inspector also has tools to debug JavaScript code, inspect files, run performances audit, ... We will describe some of these later in the book.

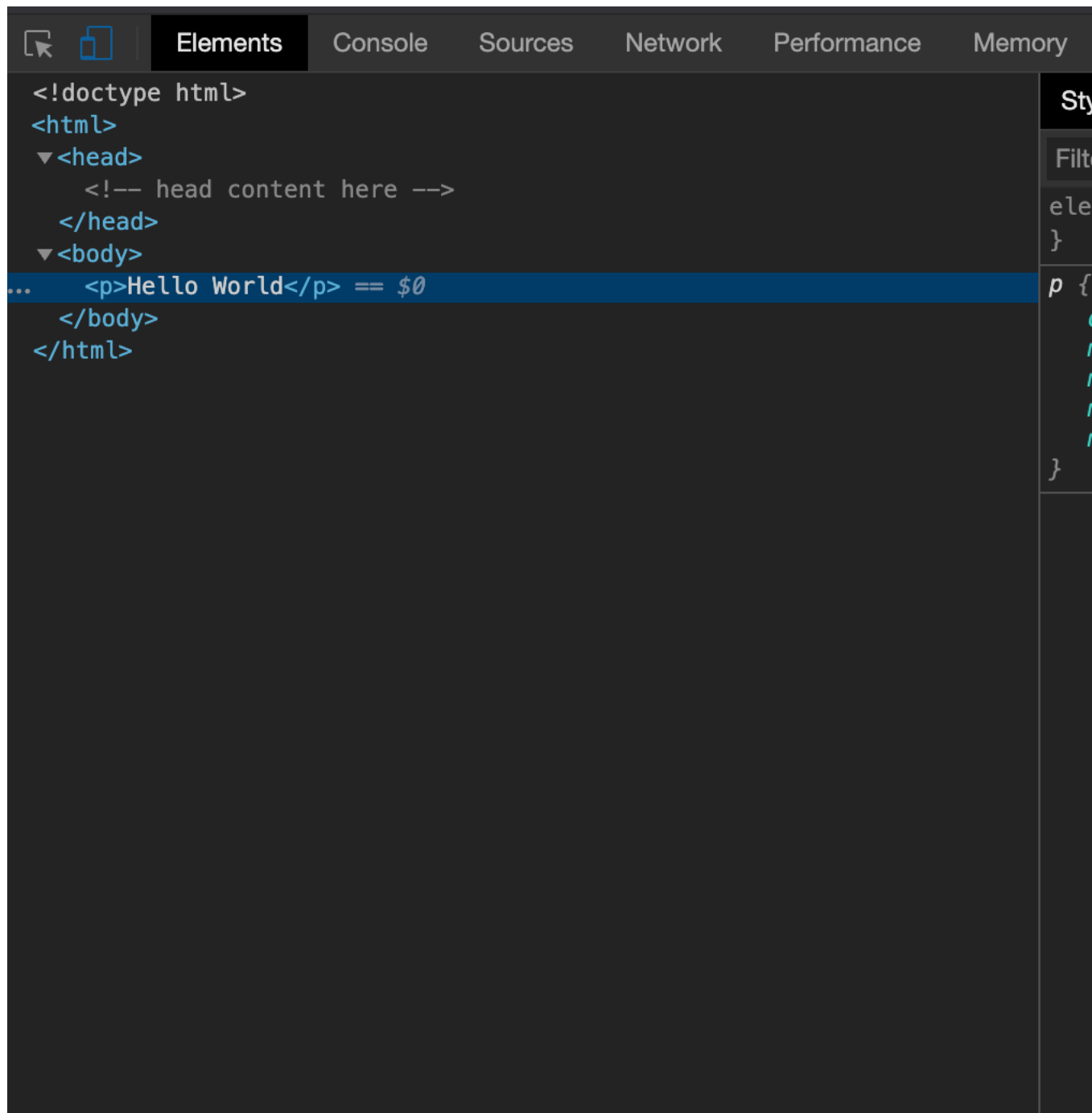


Figure 2.1: Inspection of the DOM in the Hello World example

```
p {  
  color: red;  
}
```

To include CSS in an HTML page, we use the `<style>` tag as follows:

```
<!DOCTYPE HTML>  
<html>  
  <head>  
    <style type="text/css">  
      p {  
        color: red;  
      }  
    </style>  
  </head>  
  <body>  
    <p>Hello World</p>  
  </body>  
</html>
```

You may update the `hello-world.html` script and run it in your web-browser to see the difference (this is not super impressive but a good start). There exist other ways to include CSS (see next chapters).

## 2.5.2 HTML and JavaScript

JavaScript is also going to be one of our best friend in this book. You will see how quickly/seamlessly you may add awesome feature to your shiny app.

Let's consider an example below:

```
<!DOCTYPE HTML>  
<html>  
  <head>  
    <style type="text/css">  
      p {  
        color: red;  
      }  
    </style>  
    <script language="javascript">  
      // displays an alert  
      alert('Click on the Hello World text!');  
      // change text color  
      function changeColor(color){
```

```
        document.getElementById('hello').style.color = "green";
    }
</script>
</head>
<body>
    <!-- onclick attributes applies the JavaScript function changeColor define above -->
    <p id="hello" onclick="changeColor('green')">Hello World</p>
</body>
</html>
```

In few lines of code, you can change the color of the text. Wonderful isn't it?  
Let's move to the next chapter to discover JavaScript!



## Chapter 3

# JavaScript

### 3.1 Introduction

JavaScript (JS) was created in 1995 by Brendan Eich and also known as ECMAScript (ES). Interestingly, you might have heard about ActionScript, which is no more than an implementation of ES by Adobe Systems. Nowadays, JavaScript is a centerpiece of the web and included in almost all websites.

Let's make a little experiment. If you have a personal blog (it is very popular in the RStats community) you probably know Hugo or Jekyll. These tools allow to quickly setup professional looking (or at least not too ugly) blogs in literally few minutes. You can focus on the content and this is what matters! Now, if you open the HTML inspector introduced in Chapter 2, click on the elements tab (in theory it is the first tab and open by default), and uncollapse the `<head>` tag, you see that a lot of scripts are included, as shown in Figure 3.1. Same remark in the `<body>` tag.

There are 2 main ways to include scripts:

- Use the `<script>` tag with the JS code inside
- Import an external file containing the JS code and only

```
<script type="text/javascript">  
// JS code here  
</script>
```

```
<!-- We use the src attribute to link the external file -->  
<script type="text/javascript" src="file.js">
```

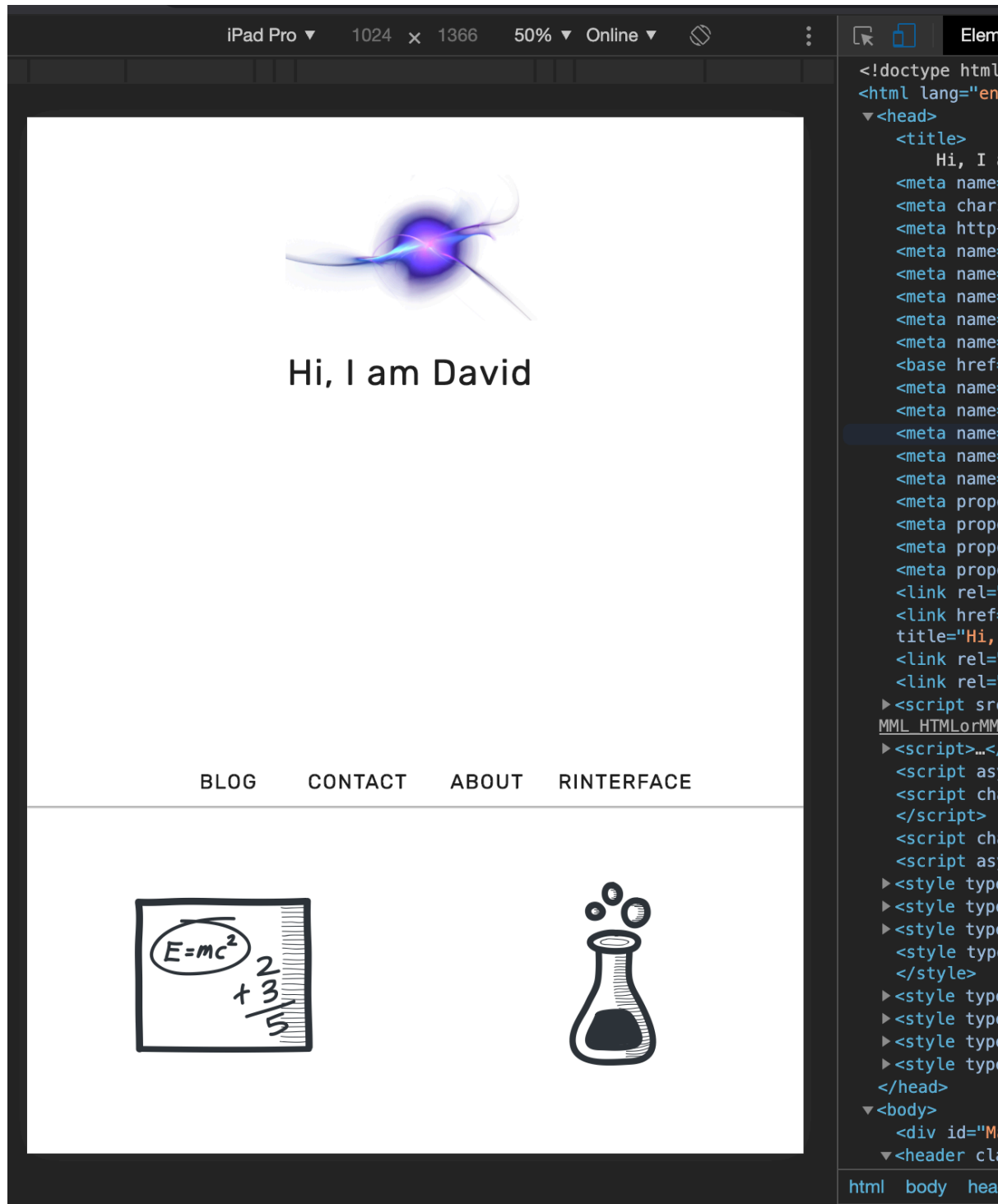


Figure 3.1: A website is full of JavaScript



Whether to choose the first or second method depends on the content of your script. If we consider jQuery, a well known JS library, it contains so much lines of code that it does not make sense to select the first method.

## 3.2 Setup

Like R or Python, JavaScript is an interpreted language. It is also executed client side, that is in the navigator. It also means that you cannot run js code without suitable tools.

### 3.2.1 Node

Node contains an interpreter for JS as well as a dependencies manager, npm (Node Package Manager). To install Node on your computer, browse to the website and follow the instruction. Once done, open a terminal and check if

```
$ which node
$ node --version
```

returns something. If not, it means that Node is not properly installed.

### 3.2.2 Choose a good IDE

I really like VSCode for all the JS things since it contains a Node interpreter and you can seamlessly execute any JS code (the truth is because I'm a big fan of the dracula color theme). But the Rstudio IDE may also be fine, provided that you have Node installed. Below, we will see how to run a JS code in both IDE.

### 3.2.3 First Script

Let's write our first script:

```
console.log("Hello World");
```

You notice that all instruction end by ;. You can run this script either in Rstudio IDE or VSCode.

In VSCode, clicking on the run arrow (top center) of Figure 3.2, triggers the `node hello.js` command, which tells Node to run my script. We see the result in the right panel (code=0 means the execution is fine and we even have the

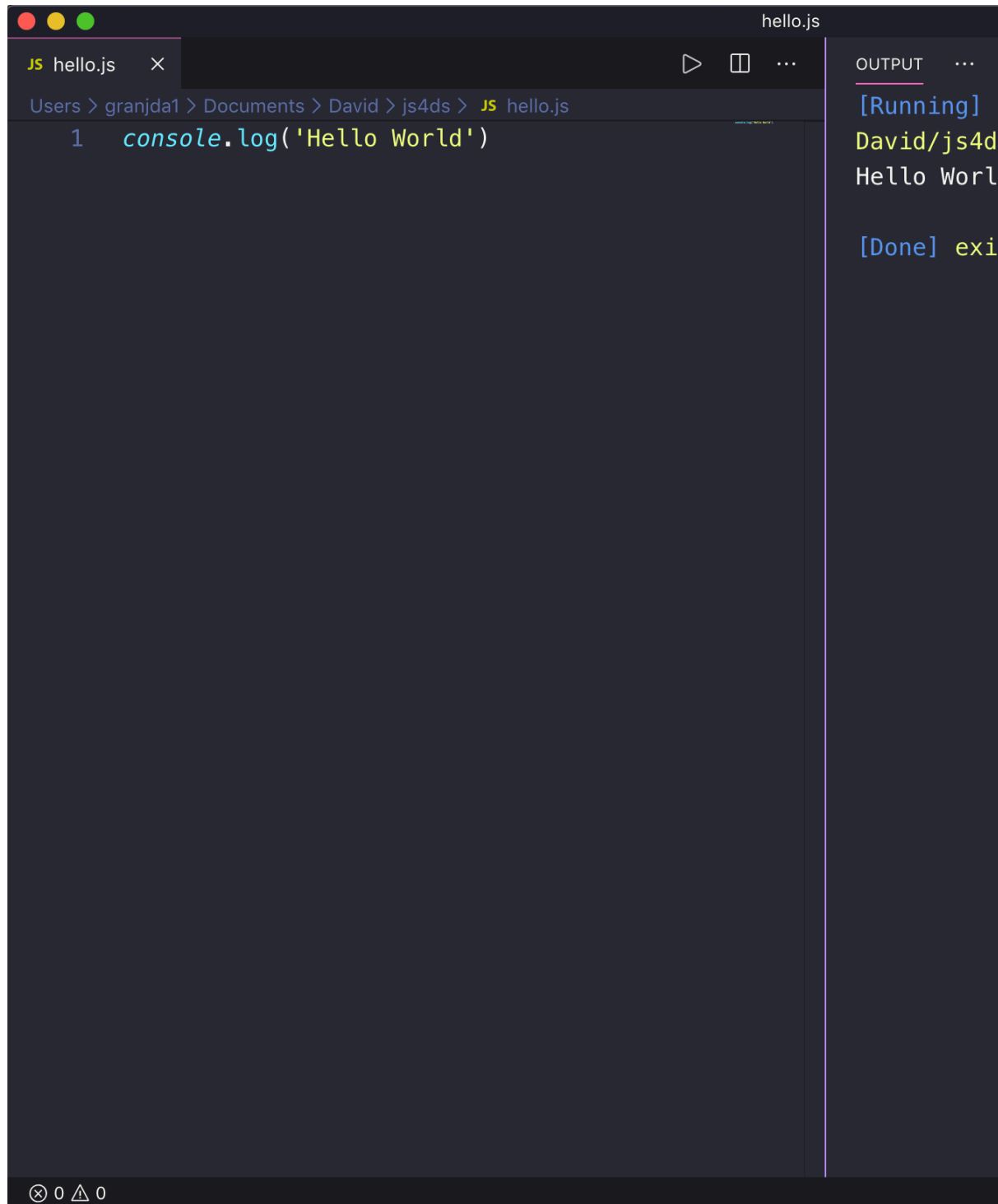
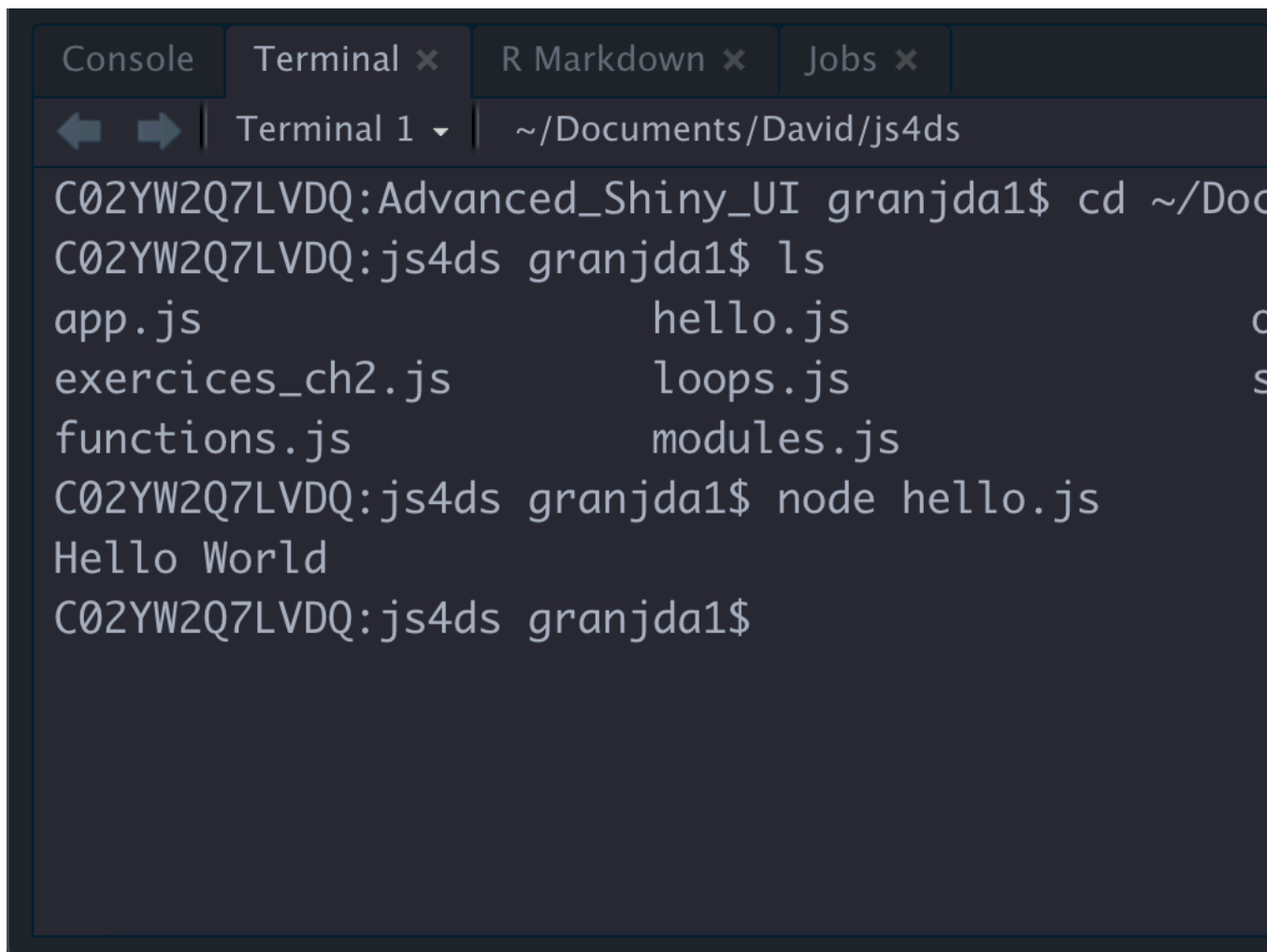


Figure 3.2: Run JS in VSCode

compute time). To run this script in the RStudio IDE, you need to click on the terminal tab (you could also open a basic terminal) and type `node hello.js` (or `node mycustompath/hello.js` if you are not in the folder containing the script). You should see the Hello World message in the console (see Figure 3.3).

A screenshot of the RStudio IDE's terminal window. The terminal has tabs for 'Console', 'Terminal', 'R Markdown', and 'Jobs'. The 'Terminal' tab is active, showing 'Terminal 1' with the path '~/Documents/David/js4ds'. The terminal output shows the following commands and results:

```
C02YW2Q7LVDQ:Advanced_Shiny_UI granjda1$ cd ~/Doc
C02YW2Q7LVDQ:js4ds granjda1$ ls
app.js                hello.js
exercices_ch2.js      loops.js
functions.js          modules.js
C02YW2Q7LVDQ:js4ds granjda1$ node hello.js
Hello World
C02YW2Q7LVDQ:js4ds granjda1$
```

Figure 3.3: Run JS in a terminal

### 3.3 Programming with JS: basis

We are now all set to introduce the basis of JS. As many languages, JS is made of variables and instructions (We saw above that instructions end by `;`).

### 3.3.1 JS types

JS defines several types:

- Number: does not distinguish between integers and others (in R for instance, numeric contains integers and double)
- String: characters ('blabla')
- Boolean: true/false

To check the type of an element, we may use the `typeof` operator (this is not a function like the `typeof` function in R).

```
typeof 1; // number
typeof 'pouic'; // string
```

### 3.3.2 Variables

A variable is defined by:

- a type
- a name
- a value

Valid variable names:

don't use an existing name like `typeof`

son't start with a number (123)

don't include any space (total price)

Based on the above forbidden items, you can use the camelCase syntax to write your variables in JS. To set a variable we use `let` (there exists `var` but this is not the latest JS norm (ES6). You will see later that we still use `var` in the shiny core and many other R packages):

```
let myVariable = 'welcome';
console.log(myVariable);
```

Then we can use all mathematical operators to manipulate a variable.

```
let myNumber = 1; // affectation
myNumber--; // decrement
console.log(myNumber); // print 0
```

List of numerical operators in JS:

+

-

\*

/

% (modulo)

++ (incrementation)

-- (decrementation)

To concatenate 2 strings, use +.

### 3.3.3 Conditions

Below are the operators to check conditions.

== (A equal B)

!= (A not equal to B)

> (>=)

< (<=)

AND (A AND B)

OR (A OR B)

To test conditions there exists several ways:

- `if (condition) { console.log('Test passed'); }`
- `if (condition) { instruction A} else { instruction B }`

This is very common to other languages (and R for instance). Whenever a lot of possible conditions need to be evaluated, it is better to choose the **switch**.

```
switch (variable) {  
  case val1: // instruction 1  
    break; // don't forget the break!  
  case val2: // instruction 2  
    break;  
  default: // when none of val1 and val2 are satisfied  
}
```

### 3.3.4 Iterations

Iterations allow to repeat an instruction or a set of instructions multiple times.

#### 3.3.4.1 For loops

The for loop has multiple ways to be used. Below is the most classic. We start by defining the index (variable). We then set an upper bound and we finish by incrementing the index value. We execute the instruction between curly braces.

```
const max = 10; // we never mentionned constants before. This is the way to call them
for (let i = 0; i <= max; i++) {
  console.log(i); // this will print i 10 times
}
```

Contrary to R, JavaScript index starts from 0 (not from 1)! This is good to keep in mind when we will mix both R and JS.

Let's have a look at the `forEach` method for arrays (introduced in ES5):

```
let letters = ["a", "b", "c", "d"];
letters.forEach((letter) => {
  console.log(letter);
});
```

Below is another way to create a for loop (introduced in ES6):

```
let samples = ['blabla', 1, null]; // this is an array!
for (let sample of samples) {
  console.log(sample);
}
```

What is the best for loop? The answer is: it depends on the situation! Actually, there even exists other ways (replace of `by` in and you get the indexes of the array, like the with the first code, but this is really not recommended).

#### 3.3.4.2 Other iterations: while

We will clearly never use them in this book but this is good to know.

```
const h = 3; i = 0;
while (i <= h) {
  console.log(i);
  i++; // we need to increment to avoid infinite loop
}
```

### 3.3.5 Objects

JavaScript is an object oriented programming language (like Python). An object is defined by:

- a type
- some properties
- some methods (to manipulate properties)

Let's define our first object below:

```
const me = {
  name : 'Divad',
  age : 29,
  music : '',
  printName: function() {
    console.log(`I am ${this.name}`);
  }
}

console.log(JSON.stringify(me)); // print a human readable object.

console.log(me.name);
console.log(me.age);
console.log(me.music);
// don't repeat yourself!!!
for (let key in me) { // here is it ok to use `in`
  console.log(`me[${key}] is ${me[key]}`);
}

me.printName();
```

Some comments on the above code:

- to access an object property, we use `object.property`
- to print a human readable version of the object `JSON.stringify` will do the job
- we introduced string interpolation with `${*}`. `*` may be any valid expression.
- methods are accessed like properties (we may also pass parameters). We use `this` to refer to the object itself. Take note, we will see it a lot!

In JavaScript, we can find already predefined objects to interact with arrays, dates.

### 3.3.5.1 Arrays

An array is a structure allowing to store informations for instance

```
let table = [1, 'plop'];
console.log(table);
```

Array may be nested

```
let nested = [1, ['a', [1, 2, 3]], 'plop'];
console.log(nested);
```

In arrays, elements may be accessed by their index, but as mentionned before, the first index is 0 (not 1 like in R). A convenient way to print all arrays's elements is to use an iteration:

```
let nested = [1, ['a', [1, 2, 3]], 'plop'];
for (let i of nested) {
  console.log(i);
}

// or with the classic approach
for (let i = 0; i < nested.length; i++) {
  console.log(nested[i]);
}
```

Note that the `length` method returns the size of an array and is very convenient in for loops. Below is a table referencing the principal methods for arrays (we will use some of them later)

Method/Property	Description
<code>length</code>	Return the number of elements in an array
<code>Join(string separator)</code>	Transform an array in a string
<code>concat(array1, array2)</code>	Assemble 2 arrays
<code>pop()</code>	Remove the last element of an array
<code>shift()</code>	Remove the first element of an array
<code>unshift(el1, el2, ...)</code>	Insert elements at the beginning of an array
<code>push(el1, el2, ...)</code>	Add extra elements at the end of an array
<code>sort()</code>	Sort array elements by increasing value of alphabetical order
<code>reverse()</code>	Symetric of <code>sort()</code>

Quite honestly, we will only use `push` and `length` in the next chapters.



### 3.3.5.2 Strings

Below are the main methods related to the String object (character in R)

Method/Property/Operator	Description
+	String concatenation
length	String length
indexOf()	Gives the position of the character following the input string
toLowerCase()	Put the string in small letters
toUpperCase()	Put the string in capital letters

### 3.3.5.3 Math

Below we mention some useful methods to handle mathematical objects

Method	Description
parseInt()	Convert a string to integer
parseFloat()	Conversion to floating number

All classic functions like `sqrt`, trigonometric functions are of course available. We call them with the `Math.*` prefix.

### 3.3.6 Functions

Functions are useful to wrap a succession of instructions to accomplish a given task. Defining functions allows programmers to save time (less copy and paste, less search and replace), do less errors and share code more easily. In modern JavaScript (ES6), functions are defined as follows:

```
const a = 1;
const fun = (parm1, parm2) => {
  console.log(a);
  let p = 3;
  return Math.max(parm1, parm2); // I use the Math object that contains the max method
}
let res = fun(1, 2);
console.log(res); // prints a and 2. a global
console.log(p); // fails because p was defined inside the function
```

This functions computes the maximum of 2 provided numbers. Some comments about scoping rules: variables defined inside the function are available for the

function but not outside. The function may use global variables defined outside of it.

### 3.3.6.1 Export functions: about modules

What happens if you wrote 100 functions and you would like to reuse some of them in different scripts? To prevent copy and pasting, we will introduce modules. Let's save the below function in a script `utils.js`:

```
const findMax = (parm1, parm2) => {  
  return Math.max(parm1, parm2); // I use the Math object that contains the max method  
}  
  
module.exports = {  
  findMax: findMax  
}
```

Now, if we create a `test.js` script in the same folder and want to use `findMax`, we need to import the corresponding module:

```
const {findMax} = require('./utils.js');  
findMax(1, 2); // prints 2
```

In the next chapters, we will see that some of the underlying JS code to build custom shiny inputs share the same utils functions. Therefore, introducing modules is necessary.

### 3.3.7 Event listeners

When you explore a web application, clicking on a button usually triggers something like a computation, a modal or an alert. How does this work? In JavaScript, interactivity plays a critical role. Indeed, you want the web application to react to user inputs like mouse clicks, keyboard events. Below we introduce DOM events.

Let's consider a basic HTML button.

```
<button id="mybutton">Go!</button>
```

On the JavaScript side, we first capture the button element using its id selector (`getElementById`).

```
const btn = document.getElementById('mybutton');
```

We then apply the `addEventListener` method. In short, an event listener is a program that triggers when a given event occurs (we can add multiple event listeners per HTML element). It takes 2 main parameters:

- the event: click, change, mouseover, ...
- the function to call

```
btn.addEventListener('click', function() {  
  alert('Thanks!');  
});
```

We could compare the JavaScript events to Shiny `observeEvent` in which we are listening to a specific user input.

## 3.4 jQuery

### 3.4.1 Introduction

jQuery is a famous JavaScript library providing a user friendly interface to manipulate the DOM and is present in almost all actual websites. It is slightly easier (understand more convenient to use) than vanilla JS, even though web developers tend to avoid it to go back to vanilla JS (Bootstrap 5, the next iteration of Bootstrap will not rely on jQuery anymore). To use jQuery in a webpage, we must include its code either by downloading the code and putting the minified JS file in our HTML or setting a link to a CDN.

```
<!doctype html>  
<html lang="en">  
<head>  
  <meta charset="utf-8">  
  <title>Including jQuery</title>  
  <!-- How to include jQuery -->  
  <script src="https://code.jquery.com/jquery-3.5.0.js"></script>  
</head>  
<body>  
  
<p>Hello World</p>  
  
<script>
```

```

$('p').css('color', 'red');
</script>

</body>
</html>

```

### 3.4.2 Syntax

Below is a minimal jQuery code representing its philosophy (“write less, do more.”):

```
$(selector).action();
```

The selector slot stands for any jQuery selector like class, id, element, [attribute], :input (will select all input elements) and many more. As a reminder, let’s consider the following example:

```
<p class="text">Hello World</p>
```

To select and interact with this element, we use JavaScript and jQuery:

```

let inner = document.getElementsByClassName('text').innerHTML; // vanilla JS
let inner = $('text').html(); // jQuery

```

This is of course possible to chain selectors

```

<ul class="list">
  <li class="item">1</li>
  <li class="item">2</li>
  <li class="item">3</li>
  <li class="item" id="precious-item">4</li>
</ul>

<ul class="list" id="list2">
  <li class="item">1</li>
  <li class="item">2</li>
  <li class="item">3</li>
  <li class="item">4</li>
</ul>

```

```

let items = $('.list .item'); // will return an array containing 8 li tags
let otherItems = $('#list2 .item'); // will select only li tags from the second ul element
let lists = $('ul'); // will return an array with 2 ul elements
let firstItem = $('#list2:first-child'); // will return the first li element of the second ul

```

jQuery is obviously simpler than pure JavaScript.

### 3.4.3 Useful functions

There exist filtering functions dedicated to simplify item selection. We gathered the one mostly used in Shiny below.

#### 3.4.3.1 Travel in the DOM

Method	Description
<code>children()</code>	Get the children of each element passed in the selector (important: only travels a single level down the DOM tree)
<code>first()</code>	Given an list of elements, select the first item
<code>last()</code>	Given an list of elements, select the last item
<code>find()</code>	Look for a descendant of the selected element(s) that could be multiple levels down in the DOM
<code>closest()</code>	Returns the first ancestor matching the condition (travels up in the DOM)
<code>filter()</code>	Fine tune element selection by applying a filter. Only return element for which the condition is true
<code>siblings()</code>	Get all siblings of the selected element(s)
<code>next()</code>	Get the immediately following sibling
<code>prev()</code>	Get the immediately preceding sibling
<code>not()</code>	Given an existing set of selected elements, remove element(s) that match the given condition

#### 3.4.3.2 Manipulate tags

Below is a list of the main jQuery methods to manipulate tags (adding class, css property...)

Method	Description
<code>addClass()</code>	Add class or multiple classes to the set of matched elements
<code>hasClass()</code>	Check if the matched element(s) have a given class

Method	Description
<code>removeClass()</code>	Remove class or multiple classes to the set of matched elements
<code>attr()</code>	Get or set the value of a specific attribute
<code>after()</code>	Insert content after
<code>before()</code>	Insert content before
<code>css()</code>	Get or set a css property
<code>remove()</code>	Remove element(s) from the DOM
<code>val()</code>	Get the current value of the matched element(s)

TO DO: add more methods

### 3.4.4 Chaining jQuery methods

A lot of jQuery methods may be chained, that is like pipe operations in R.

```
<ul>
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
  <li>Item 4</li>
  <li>Item 5</li>
</ul>
```

We end the chain by ; and each step is indent by 2 spaces in the right direction.

```
$('#ul')
  .first()
  .css('color', 'green') // add some style with css
  .attr('id', 'myAwesomeItem') // add an id attribute
  .addClass('amazing-ul');
```

### 3.4.5 Iterations

Like in vanilla JavaScript, it is possible to do iterations in jQuery. Let's consider the following HTML elements.

```
<ul>
  <li>Item 1</li>
  <li>Item 2</li>
</ul>
```

We apply the `each` method to change the style of each matched element step by step.

```
$('li').each(function() {  
    $(this).css('visibility', 'hidden'); // will hide all li items  
});
```

The `map` method has a different purpose. It creates a new object based on the provided one.

```
let items = [0,1,2,3,4,5];  
let threshold = 3;  
  
let filteredItems = $(items).map(function(i) {  
    // removes all items > threshold  
    if (i > threshold)  
        return null;  
    return i;  
});
```

### 3.4.6 Good practice

It is recommended to wrap any jQuery code as follows:

```
$(document).ready(function(){  
    // your code  
});  
  
// or a shortcut  
  
$(function() {  
    // your code  
});
```

Indeed, do you guess what would happen if you try to modify an element that does not even exist? The code above will make sure that the document is ready before starting any jQuery manipulation.

### 3.4.7 Events

In jQuery there exists a significant number of methods related to events. Below are the most popular:

```

$(element).click(); // click event
$(element).change(); // trigger change on an element
$(element).on('click', function() {
  // whatever
}); // attach an event handler function. Here we add click for the example
$(element).one('click', function() {
  // whatever
}); // the difference with on is that one will trigger only once
$(element).resize(); // useful to trigger plot resize in Shiny so that they correctly .
$(element).trigger('change') // similar to $(element).change(); You will find it in th

```

The `.on` event is frequently used in Shiny since it allows to pass custom events which are not part of the JS predefined events. For instance shinydashboard relies on a specific HTML/JavaScript/CSS template including a homemade API for handling the dashboard events. Don't worry if this section is not clear at the moment. We will see practical examples in the following chapters.

### 3.4.8 Extending objects

A last feature we need to mention about jQuery is the ability to extend objects with additional properties and/or method.

```

// jQuery way
$(function() {
  let object1 = {
    apple: 0
  };
  $.extend(object1, {
    print: function() {
      console.log(this);
    }
  });
  object1.print();
});

```

With vanilla JS we would use `Object.defineProperty`:

```

// pure JavaScript
Object.defineProperty(object1, 'print', {
  value: function() {
    console.log(this);
  },
  writable: false
});

```



## Chapter 4

# Shiny, under the hood

In the 2 previous chapters, we quickly introduced HTML and JavaScript. Chapter 1 finished on an example showing how to modify an HTML page with JavaScript. Yet, in Chapter 2, we simply introduced the language without showing any link with HTML. In this chapter we are going to see what Shiny has under the hood to better understand the link between those languages. We will particularly accord a great importance to jQuery.

We will answer to the following questions:

- What web dependencies is Shiny based on?
- How does Shiny deal with inputs?
- How is R/JavaScript communication achieved?

### 4.1 Shiny dependencies

This book assumes you are already quite advanced in Shiny. Below we will investigate elements that you probably never noticed.

Shiny allows to develop web applications by only using R. If you remember about the first experiment of Chapter 2, we only did

```
library(shiny)
p("Hello World")
```

to notice that the `p` function generates HTML. We will see in the next chapters other tools to build/modify/delete tags. The main difference between HTML tags and Shiny tags is the absence of closing tag for Shiny. For instance, in raw HTML, we expect `<p>` to be closed by `</p>`. In Shiny, we only call `p(...)`,

where ... may be attributes like class/id or children tags. However, this is still a bit far from web application since there is no user interface, interactivity and computations. The simplest Shiny layout is the `fluidPage` (if you type `shinyapp` in the R console, it will show a predefined snippet with this default template):

```
ui <- fluidPage(  
  p("Hello World")  
)  
  
server <- function(input, output, session) {}  
shinyApp(ui, server)
```

At first glance, the page only contains the Hello World text. Waiiit ... are you sure about this? Let's run the above example and open the HTML inspector. Results are displayed on Figure 4.1. In Chapter 6 we will see better tools to extract HTML dependencies.

We see in the head section that Shiny has 4 dependencies:

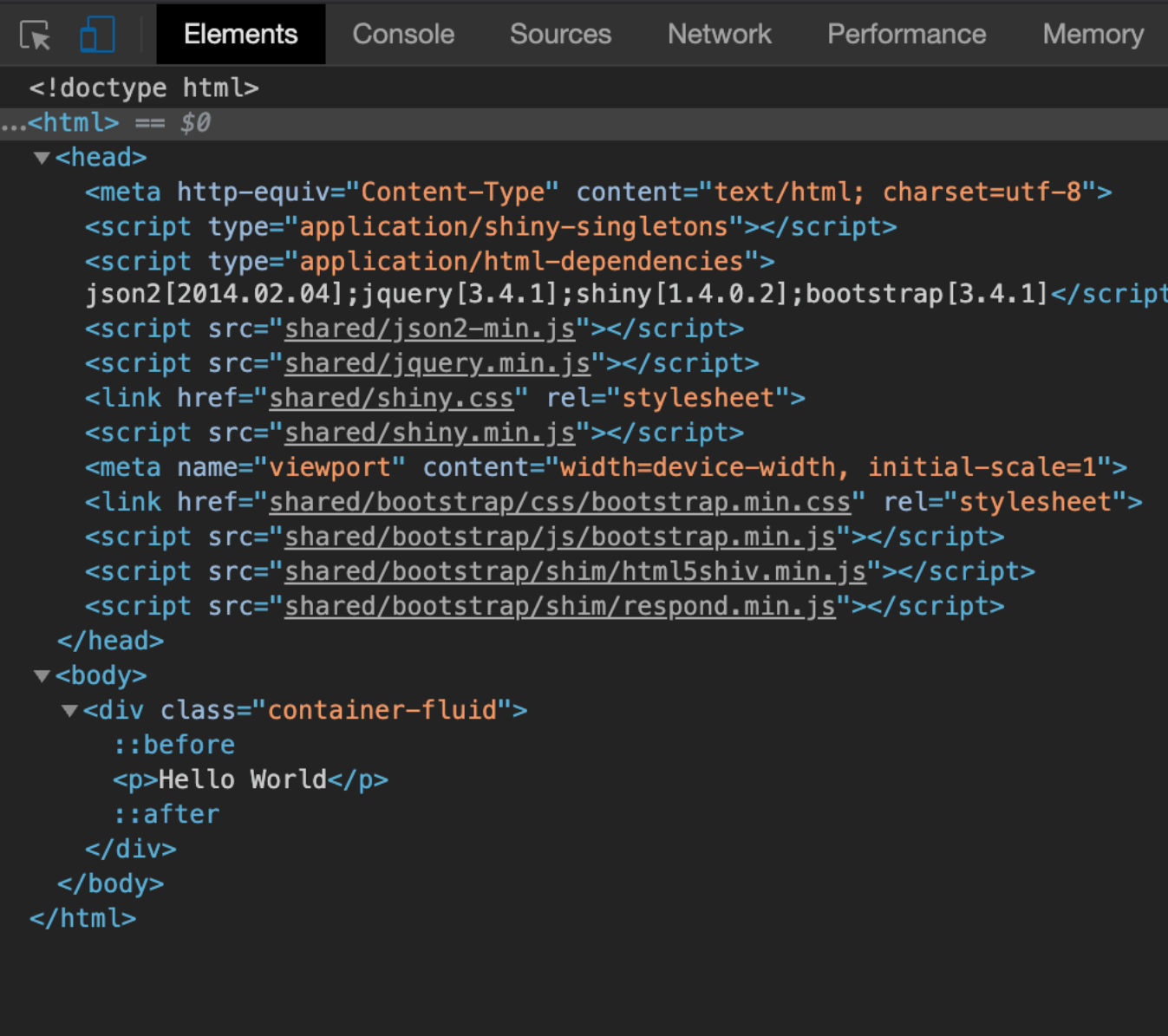
- json2
- jQuery 3.4.1
- shiny (custom JavaScript and CSS)
- Bootstrap 3.4.1 (JavaScript and CSS) + other files (html5shiv, respond)

Bootstrap is here to provide plug and play design and interactions (tabs, navs). For instance the `fluidRow` and `column` functions of Shiny leverage the Bootstrap grid to control how elements are displayed in a page. This is convenient because it avoids to write a crazy amount of CSS/JavaScript and always reinvent the wheel. jQuery drives the DOM manipulations. Shiny has its own JS and CSS files (we will discuss this very soon). Finally, json2 is a library to handle the JSON data format (JavaScript Object Notation). In the following chapters we will use it a lot, through the `jsonlite` package that allows to transform JSON objects in R objects and inversely.

In summary, all those libraries are necessary to make Shiny what it is! Customizing Shiny will imply to alter those existing libraries (except the Shiny core JavaScript and json2).

## 4.2 Shiny hiddens gems

As promised earlier, let's talk about the Shiny JavaScript core. The goal of this part is to better understand the mechanisms behind Shiny, especially the input system.



```
<!doctype html>
...<html> == $0
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <script type="application/shiny-singletons"></script>
    <script type="application/html-dependencies">
      json2[2014.02.04];jquery[3.4.1];shiny[1.4.0.2];bootstrap[3.4.1]</script>
    <script src="shared/json2-min.js"></script>
    <script src="shared/jquery.min.js"></script>
    <link href="shared/shiny.css" rel="stylesheet">
    <script src="shared/shiny.min.js"></script>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link href="shared/bootstrap/css/bootstrap.min.css" rel="stylesheet">
    <script src="shared/bootstrap/js/bootstrap.min.js"></script>
    <script src="shared/bootstrap/shim/html5shiv.min.js"></script>
    <script src="shared/bootstrap/shim/respond.min.js"></script>
  </head>
  <body>
    <div class="container-fluid">
      ::before
      <p>Hello World</p>
      ::after
    </div>
  </body>
</html>
```

Figure 4.1: Shiny dependencies

### 4.2.1 Initialization

Upon initialization, Shiny runs several JavaScript functions. Not surprisingly, there is one called `initShiny` containing a substantial amount of elements.

We find utils functions like `bindOutputs`, `unbindOutputs` to respectively bind/unbind outputs, `bindInputs` and `unbindInputs` for inputs. Only `bindAll` and `unbindAll` are available to the user (see a usecase here). To illustrate what they do, let's run the app below.

```
ui <- fluidPage(  
  sliderInput("obs", "Number of observations:",  
    min = 0, max = 1000, value = 500  
  ),  
  plotOutput("distPlot")  
)  
  
server <- function(input, output, session) {  
  output$distPlot <- renderPlot({  
    hist(rnorm(input$obs))  
  })  
}  
shinyApp(ui, server)
```

We then open the HTML inspector and run `Shiny.unbindAll(document)` (document is the scope, that is where to search). Try to change the slider input. What do you observe? Now let's type `Shiny.bindAll(document)` and update the slider value. What happens? Magic isn't it? This simply shows that when inputs are not bound, nothing happens so binding inputs is necessary. Let's see below how an input binding works.

### 4.2.2 Input bindings

The input binding is defined by a class living in the `input_binding.js` file. An input binding allows Shiny to identify each instance of a given input and what you can do with this input. The interesting thing is that if your app contains 10 different sliders, they all share the same input binding! An input binding is an object having the following methods (but not only):

- `find(scope)`: this method specifies how to find the current input element (el) in the DOM. scope refers to the `document` element. In general, we use jQuery selector to search for a class.
- `initialize(el)`: This is called before the input is bound but not all input need to be initialized. Some API like Framework7 require to almost always have an initialize method (We will see later).

- `getValue(el)`: returns the input value. The way to obtain the value tightly depends on the object and is different for almost all inputs.
- `setValue(el, value)`: This method is used to set the value of the current input.
- `receiveMessage(el, data)`: This method is the JavaScript part of all the `updateInput` functions. We usually call the `setValue` method inside.
- `subscribe(el, callback)`: We listen to events telling under which circumstances to change the input value. Some API like Bootstrap explicitly mention those events (like `hide.bs.tab`, `shown.bs.tab`, ...).
- `getRatePolicy`: when `callback` is true in the `subscribe` method, we apply a specific rate policy (debounce, throttle). This is useful for instance when we don't want to flood the server with useless update requests. For a slider, we only want to send the value as soon as the range stops moving and not all intermediate values. Those elements are defined here.

At the end of the input binding definition, we register it for Shiny.

```
let myBinding = new Shiny.inputBinding();
$.extend(myBinding, {
  // methods go here
});

Shiny.inputBindings.register(myBinding, 'reference');
```

Although the Shiny documentation mentions a `Shiny.inputBindings.setPriority` method to handle conflicting bindings, it is better not to use it.

Upon initialization, Shiny calls the `initializeInputs` function that takes all input bindings and call their `initialize` method before binding all inputs. Note that once an input has been initialized it has a `_shiny_initialized` tag to avoid initializing it twice. As shown above, the `initialize` method is not always defined.

TO DO: picture to add visual representation of an input binding

### 4.2.3 websocket

How does R (server) and JavaScript (client) communicate? This is a builtin Shiny feature highlighted here, leveraging the `httpuv` and `websocket` packages. We will not detail how they work but rather how to inspect the websocket in a web browser. Let's run the following app.

```
shinyApp(
  ui = fluidPage(
    selectInput("variable", "Variable:",
```

```

        c("Cylinders" = "cyl",
          "Transmission" = "am",
          "Gears" = "gear")),
      tableOutput("data")
    ),
    server = function(input, output) {
      output$data <- renderTable({
        mtcars[, c("mpg", input$variable), drop = FALSE]
      }, rownames = TRUE)
    }
  )
)

```

After opening the HTML inspector, we select the network tab and search for websocket in the list. We also choose the message tab to inspect what R and JavaScript say to each others. On the JavaScript side, the websocket is created in the shinyapp.js file. The first element received from R is the first message in the list shown in Figure 4.2. It is a JSON containing the method used as well as passed data. In the meantime, you may change the select input value.

```

socket.send(JSON.stringify({
  method: 'init',
  data: self.$initialInput
}));

```

The second message received from R is after updating the select input.

```

this.sendInput = function(values) {
  var msg = JSON.stringify({
    method: 'update',
    data: values
  });

  // other things
};

```

All of this is quite complex but extremely useful to check whether input/output work properly. In case of error, we would see the field **error** containing some elements. In the last part of this book, we will be designing custom inputs and knowing how to debug them outside R is priceless.

Finally, `Shiny.shinyapp.$socket.readyState` returns the state of the socket connection. It should be 1 if your app is running but I've seen some cases where the socket was actually closed (and nothing could happen).

The screenshot shows a web browser window displaying a Shiny application. The browser's address bar shows the URL `127.0.0.1:6087`. The application interface includes a "Variable:" dropdown menu set to "Transmission" and a table of car data with columns "mpg" and "am".

	mpg	am
Mazda RX4	21.00	1.00
Mazda RX4 Wag	21.00	1.00
Datsun 710	22.80	1.00
Hornet 4 Drive	21.40	0.00
Hornet Sportabout	18.70	0.00
Valiant	18.10	0.00
Duster 360	14.30	0.00
Merc 240D	24.40	0.00
Merc 230	22.80	0.00
Merc 280	19.20	0.00
Merc 280C	17.80	0.00
Merc 450SE	16.40	0.00
Merc 450SL	17.30	0.00
Merc 450SLC	15.20	0.00
Cadillac Fleetwood	10.40	0.00
Lincoln Continental	10.40	0.00
Chrysler Imperial	14.70	0.00
Fiat 128	32.40	1.00
Honda Civic	30.40	1.00
Toyota Corolla	33.90	1.00
Toyota Corona	21.50	0.00
Dodge Challenger	15.50	0.00
AMC Javelin	15.20	0.00
Camaro Z28	13.30	0.00
Pontiac Firebird	19.20	0.00
Fiat X1-9	27.30	1.00
Porsche 914-2	26.00	1.00
Lotus Europa	30.40	1.00
Ford Pantera L	15.80	1.00
Ferrari Dino	19.70	1.00
Maserati Bora	15.00	1.00
Volvo 142E	21.40	1.00

The browser's developer tools are open, showing the "Elements" panel on the right. The "Name" column lists various resources, including `127.0.0.1`, `json2-min.js`, `jquery.min.js`, `shiny.css`, `shiny.min.js`, `selectize.bootstra...`, `selectize.min.js`, `bootstrap.min.css`, `bootstrap.min.js`, `html5shiv.min.js`, `respond.min.js`, `favicon.ico`, and `websocket/`. The "Data" column shows the corresponding data for each resource, including JSON objects for initialization and configuration.

Figure 4.2: Shiny websocket

### 4.2.4 Custom handlers: from R to JavaScript

Shiny contains tools to ease the communication between R and JavaScript. This is what happens in the last part. If you remember, we were playing with a `selectInput` and a `datatable`. How does R send messages to JavaScript?

Shiny is made of input and output. Yet there is a third parameter you can pass to the server function called `session`. The `session` object contains informations on the current session like `clientData`, the namespace `ns` (if working inside modules), as well as methods (yes methods since `session` is an R6 class). Among those methods, we are interested by 2 of them:

- `sendCustomMessage(type, message)`: > Custom messages have no meaning to Shiny itself; they are used solely to convey information to custom JavaScript logic in the browser
- `sendInputMessage()`: > if the input is present and bound on the page at the time the message is received, then the input binding object's `receiveMessage(el, message)` method will be called

While `sendInputMessage` is the R side of the input features, `sendCustomMessage` simply allows to communicate between R and JavaScript. Basically a R function :

```
sayHelloToJS <- function(text, session) {
  session$sendCustomMessage(type = 'say-hello', message = text)
}
```

We need a JavaScript receptor, handle by the `addCustomMessageHandler` method:

```
Shiny.AddCustomMessageHandler('say-hello', function(message) {
  alert(`R says ${message} to you!`)
})
```

Of course, the `type` parameter is critical to make the connection between R and JavaScript and 2 different message handlers must have different types to avoid conflicts.

TO DO: picture showing the communication

### 4.2.5 Utilities to quickly define new inputs

If you ever wondered where the `Shiny.onInputChange` or `Shiny.setInputValue` comes from (see article), it is actually defined in the `initShiny` function.



```
exports.setInputValue = exports.onInputChange = function(name, value, opts) {  
  opts = addDefaultInputOpts(opts);  
  inputs.setInput(name, value, opts);  
};
```

Briefly, this function avoids to create an input binding. It is faster to code but there is a price to pay: you lose the possibility to easily update the new input. Indeed, all input functions like `sliderInput` have their own update function like `updateSliderInput`, because of the custom input binding system (We will see it very soon)!

#### 4.2.6 Get access to initial values

Something we may notice when exploring the `initShiny` function is the existence of a shinyapp object, defined as follows:

```
var shinyapp = exports.shinyapp = new ShinyApp();
```

Let's explore what `ShinyApp` contains. The definition is located in the `shinyapps.js` script.

```
var ShinyApp = function() {  
  this.$socket = null;  
  
  // Cached input values  
  this.$inputValues = {};  
  
  // Input values at initialization (and reconnect)  
  this.$initialInput = {};  
  
  // Output bindings  
  this.$bindings = {};  
  
  // Cached values/errors  
  this.$values = {};  
  this.$errors = {};  
  
  // Conditional bindings (show/hide element based on expression)  
  this.$conditionals = {};  
  
  this.$pendingMessages = [];  
  this.$activeRequests = {};  
  this.$nextRequestId = 0;
```

```
this.$allowReconnect = false;  
};
```

It creates several properties, some of them are easy to guess like `inputValues` or `initialInput`. Let's run the example below and open the HTML inspector. Notice that the `sliderInput` is set to 500 at `t0` (initialization).

```
ui <- fluidPage(  
  sliderInput("obs", "Number of observations:",  
    min = 0, max = 1000, value = 500  
  ),  
  plotOutput("distPlot")  
)  
  
server <- function(input, output, session) {  
  output$distPlot <- renderPlot({  
    hist(rnorm(input$obs))  
  })  
}  
shinyApp(ui, server)
```

Figure 4.3 shows how to access Shiny's initial input value with `Shiny.shinyapp.$initialInput.obs`. After changing the slider position, its value is given by `Shiny.shinyapp.$inputValues.obs`. `$initialInput` and `$inputValues` contains way more elements but we are only interested by the slider in this example.

I acknowledge, the practical interest might be limited but still good to know for debugging purposes.

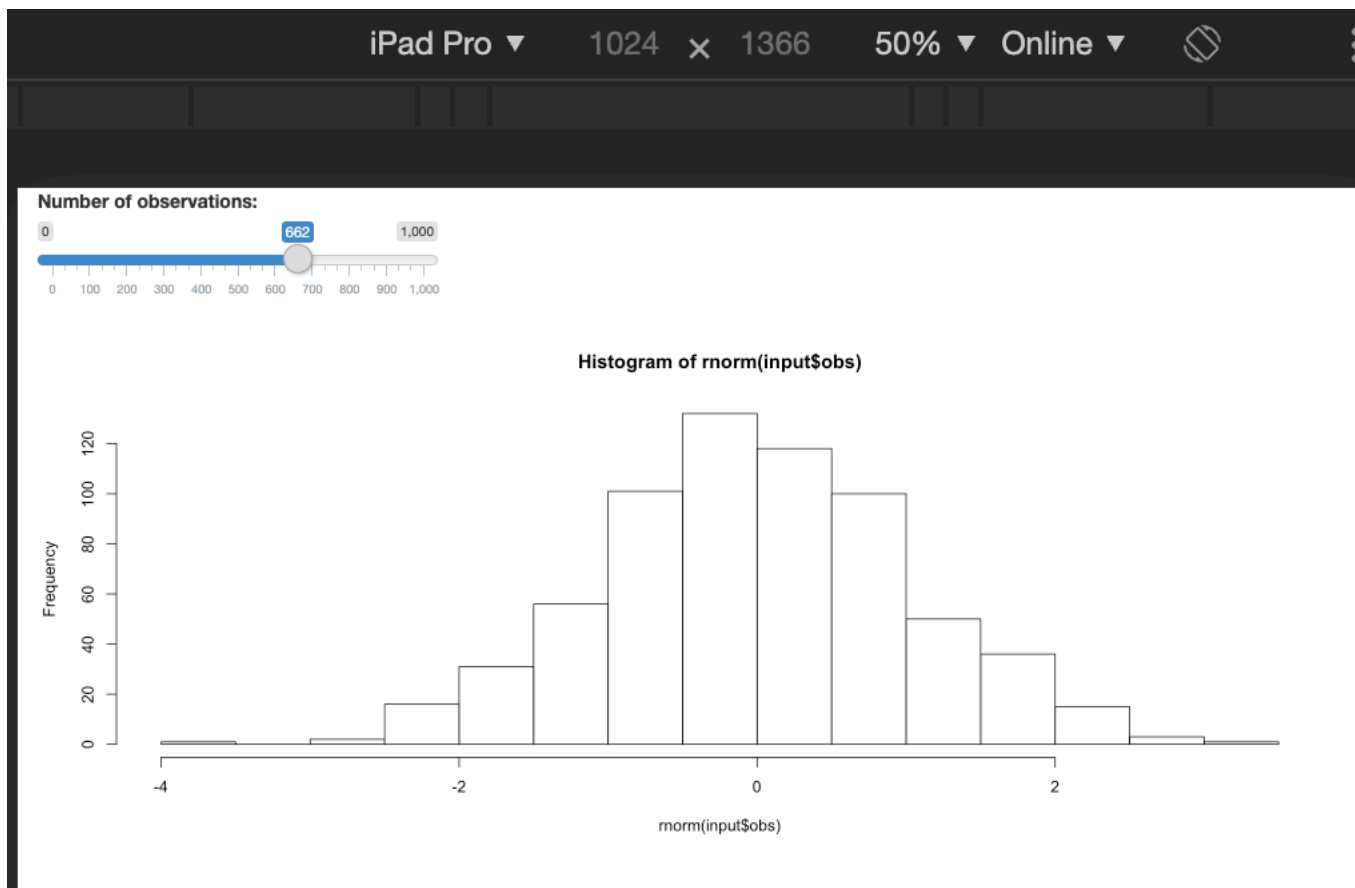


Figure 4.3: Explore initial input values



htmltools



While building a custom html template, you will need to know more about the wonderful `htmltools` developed by Winston Chang, member of the shiny core team. It has the same spirit as `devtools`, that is, making your web developer life easier. What follows does not have the pretention to be an exhaustive guide about this package. Yet, it will provide you with the main tools to be more efficient.





## Chapter 5

# htmltools overview

### 5.1 HTML Tags

htmltools contains tools to write HTML tags we saw in Chapter 2:

```
div()
```

If you had to gather multiple tags together, prefer `tagList()` as `list()`, although the HTML output is the same. The first has the `shiny.tag.list` class in addition to `list`. (The `Golem` package allows to test if a R object is a tag list, therefore using `list` would make the test fail).

### 5.2 Notations

Whether to use `tags$div` or `div` depends if the tag is exported by default. For instance, you could use `htmltools::div` but not `htmltools::nav` since `nav` does not have a dedicated function (only for `p`, `h1`, `h2`, `h3`, `h4`, `h5`, `h6`, `a`, `br`, `div`, `span`, `pre`, `code`, `img`, `strong`, `em`, `hr`). Rather use `htmltools::tags$nav`. Alternatively, there exists a function (in `shiny` and `htmltools`) called `withTags()`. Wrapping your code in this function enables you to use `withTags(nav(), ...)` instead of `tags$nav()`.

### 5.3 Adding new tags

The `tag` function allows to add extra HTML tags not already defined. You may use it as follows:

```

tag("test", list(class = "test", p("Custom Tag")))
# structure below
tag
  "test"
  list
    class = "test"
    p
      "Custom Tag"

```

## 5.4 Alternative way to write tags

htmltools comes with the `HTML()` function that you can feed with raw HTML:

```

HTML('<div>Blabla</div>')
# will render exactly like
div("Blabla")

# but there class is different
class(HTML('<div>Blabla</div>'))
class(div("Blabla"))

```

You will not be able to use tag related functions, as in the following parts. Therefore, I strongly recommend using R and not mixing HTML in R. Interestingly, if you want to convert HTML to R code, there is a Shiny App developed by Alan Dipert from RStudio, namely `html2R`. There are some issues, non standard attributes (like `data-toggle`) are not correctly processed but there are fixes. This will save you precious time!

## 5.5 Playing with tags

### 5.5.1 Tags structure

According to the `tag` function, a tag has:

- a name such as `span`, `div`, `h1` ... `tag$name`
- some attributes, which you can access with `tag$attribs`
- children, which you can access with `tag$children`
- a class, namely `"shiny.tag"`

For instance:

```
# create the tag
myTag <- div(
  class = "divclass",
  id = "first",
  h1("Here comes your baby"),
  span(class = "child", id = "baby", "Crying")
)
# access its name
myTag$name
# access its attributes (id and class)
myTag$attrs
# access children (returns a list of 2 elements)
myTag$children
# access its class
class(myTag)
```

How to modify the class of the second child, namely span?

```
second_children <- myTag$children[[2]]
second_children$attrs$class <- "adult"
myTag
# Hummm, this is not working ...
```

Why is this not working? By assigning `myTag$children[[2]]` to `second_children`, `second_children$attrs$class <- "adult"` modifies the class of the copy and not the original object. Thus we do:

```
myTag$children[[2]]$attrs$class <- "adult"
myTag
```

In the following section we explore helper functions, such as `tagAppendChild` from `htmltools`.

## 5.5.2 Useful functions for tags

`htmltools` and `Shiny` have powerful functions to easily add attributes to tags, check for existing attributes, get attributes and add other siblings to a list of tags.

### 5.5.2.1 Add attributes

- `tagAppendAttributes`: this function allow you to add a new attribute to the current tag.

For instance, assuming you created a div for which you forgot to add an id attribute:

```
mydiv <- div("Where is my brain")
mydiv <- tagAppendAttributes(mydiv, id = "here_it_is")
```

You can pass as many attributes as you want, including non standard attributes such as `data-toggle` (see Bootstrap 3 tabs for instance):

```
mydiv <- tagAppendAttributes(mydiv, `data-toggle` = "tabs")
# even though you could proceed as follows
mydiv$attribs[["aria-controls"]] <- "home"
```

### 5.5.2.2 Check if tag has specific attribute

- `tagHasAttribute`: to check if a tag has a specific attribute

```
# I want to know if div has a class
mydiv <- div(class = "myclass")
has_class <- tagHasAttribute(mydiv, "class")
has_class
# if you are familiar with %>%
has_class <- mydiv %>% tagHasAttribute("class")
has_class
```

### 5.5.2.3 Get all attributes

- `tagGetAttribute`: to get the value of the targeted attributes, if it exists, otherwise `NULL`.

```
mydiv <- div(class = "test")
# returns the class
tagGetAttribute(mydiv, "class")
# returns NULL
tagGetAttribute(mydiv, "id")
```

### 5.5.2.4 Set child/children

- `tagSetChildren` allows to create children for a given tag. For instance:

```
mydiv <- div(class = "parent", id = "mother", "Not the mama!!!")
# mydiv has 1 child "Not the mama!!!"
mydiv
children <- lapply(1:3, span)
mydiv <- tagSetChildren(mydiv, children)
# mydiv has 3 children, the first one was removed
mydiv
```

Notice that `tagSetChildren` removes all existing children. Below we see another set of functions to add children while conserving existing ones.

#### 5.5.2.5 Add child or children

- `tagAppendChild` and `tagAppendChildren`: add other tags to an existing tag. Whereas `tagAppendChild` only takes one tag, you can pass a list of tags to `tagAppendChildren`.

```
mydiv <- div(class = "parent", id = "mother", "Not the mama!!!")
otherTag <- span("I am your child")
mydiv <- tagAppendChild(mydiv, otherTag)
```

You might wonder why there is no `tagRemoveChild` or `tagRemoveAttributes`. Let's look at the `tagAppendChild`

```
tagAppendChild <- function(tag, child) {
  tag$children[[length(tag$children) + 1]] <- child
  tag
}
```

Below we write the `tagRemoveChild`, where `tag` is the target and `n` is the position to remove in the list of children:

```
mydiv <- div(class = "parent", id = "mother", "Not the mama!!!", span("Hey!"))

# we create the tagRemoveChild function
tagRemoveChild <- function(tag, n) {
  # check if the list is empty
  if (length(tag$children) == 0) {
    stop(paste(tag$name, "does not have any children!"))
  }
  tag$children[n] <- NULL
  tag
}
```

```
mydiv <- tagRemoveChild(mydiv, 1)
mydiv
```

When defining the `tagRemoveChild`, we choose `[]` instead of `[[` to allow to select multiple list elements:

```
mydiv <- div(class = "parent", id = "mother", "Not the mama!!!", "Hey!")
# fails
`[[`(mydiv$children, c(1, 2))
# works
`[`(mydiv$children, c(1, 2))
```

Alternatively, we could also create a `tagRemoveChildren` function. Also notice that the function raises an error if the provided tag does not have children.

### 5.5.3 Other interesting functions

The Golem package written by thinkr contains neat functions to edit your tags. Particularly, the `tagRemoveAttributes`:

```
tagRemoveAttributes <- function(tag, ...) {
  attrs <- as.character(list(...))
  for (i in seq_along(attrs)) {
    tag$attribs[[ attrs[i] ]] <- NULL
  }
  tag
}
```

```
mydiv <- div(class = "test", id = "coucou", "Hello")
tagRemoveAttributes(mydiv, "class", "id")
```

### 5.5.4 Conditionally set attributes

Sometimes, you only want to set attributes under specific conditions.

```
my_button <- function(color = NULL) {
  tags$button(
    style = paste("color:", color),
    p("Hello")
  )
}

my_button()
```

This example will not fail but having `style="color: "` is not clean. We may use conditions:

```
my_button <- function(color = NULL) {
  tags$button(
    style = if (!is.null(color)) paste("color:", color),
    p("Hello")
  )
}

my_button("blue")
my_button()
```

In this example, style won't be available if color is not specified.

### 5.5.5 Using %>%

While doing a lot of manipulation for a tag, if you don't need to create intermediate objects, this is a good idea to use %>% from magrittr:

```
div(class = "cl", h1("Hello")) %>%
  tagAppendAttributes(id = "myid") %>%
  tagAppendChild(p("some extra text here!"))
```

### 5.5.6 Programmatically create children elements

Assume you want to create a tag with 3 children inside:

```
div(
  span(1),
  span(2),
  span(3),
  span(4),
  span(5)
)
```

The structure is correct but imagine if you had to create 1000 `span` or fancier tag. The previous approach is not consistent with DRY programming. `lapply` function will be useful here (or the purrr `map` family):

```
# base R
div(lapply(1:5, function(i) span(i)))
# purrr + %>%
map(1:5, function(i) span(i)) %>% div()
```





## Chapter 6

# Dependency utilities

When creating a new template, you sometimes need to import custom HTML dependencies that do not come along with shiny. No problem, `htmltools` is here for you (shiny also contains these functions).

### 6.1 The dirty approach

Let's consider the following example. I want to include a bootstrap 4 card in a shiny app. This example is taken from an interesting question [here](#). The naive approach would be to include the HTML code directly in the app code

```
# we create the card function before
my_card <- function(...) {
  withTags(
    div(
      class = "card border-success mb-3",
      div(class = "card-header bg-transparent border-success"),
      div(
        class = "card-body text-success",
        h3(class = "card-title", "title"),
        p(class = "card-text", ...)
      ),
      div(class = "card-footer bg-transparent border-success", "footer")
    )
  )
}

# we build our app
shinyApp(
```

```

ui = fluidPage(
  fluidRow(
    column(
      width = 6,
      align = "center",
      br(),
      my_card("blablabla. PouetPouet Pouet.")
    )
  ),
  server = function(input, output) {}
)

```

and desperately see that nothing is displayed. If you remember, this was expected since shiny does not contain bootstrap 4 dependencies and this card is unfortunately a bootstrap 4 object. Don't panic! We just need to tell shiny to load the css we need to display this card (if required, we could include the javascript as well). We could use either `includeCSS()`, `tags$head(tags$link(rel = "stylesheet", type = "text/css", href = "custom.css"))`. See more here.

```

shinyApp(
  ui = fluidPage(
    # load the css code
    includeCSS(path = "https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.m
  fluidRow(
    column(
      width = 6,
      align = "center",
      br(),
      my_card("blablabla. PouetPouet Pouet.")
    )
  ),
  server = function(input, output) {}
)

```

The card is ugly (which is another problem we will fix later) but at least displayed.

When I say this approach is dirty, it is because it will not be easily re-usable by others. Instead, we prefer a packaging approach, like in the next section.

## 6.2 The clean approach

We will use the `htmlDependency` and `attachDependencies` functions from `htmltools`. The `htmlDependency` takes several arguments:

- the name of your dependency
- the version (useful to remember on which version it is built upon)
- a path to the dependency (can be a CDN or a local folder)
- script and stylesheet to respectively pass css and scripts

```
# handle dependency
card_css <- "bootstrap.min.css"
bs4_card_dep <- function() {
  htmlDependency(
    name = "bs4_card",
    version = "1.0",
    src = c(href = "https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/"),
    stylesheet = card_css
  )
}
```

We create the card tag and give it the bootstrap 4 dependency through the `attachDependencies()` function. In recent version of `htmltools`, we may simply use `tagList(tag, deps)` instead.

```
# create the card
my_card <- function(...) {
  cardTag <- withTags(
    div(
      class = "card border-success mb-3",
      div(class = "card-header bg-transparent border-success"),
      div(
        class = "card-body text-success",
        h3(class = "card-title", "title"),
        p(class = "card-text", ...)
      ),
      div(class = "card-footer bg-transparent border-success", "footer")
    )
  )

  # attach dependencies (old way)
  # htmltools::attachDependencies(cardTag, bs4_card_dep())

  # simpler way
  tagList(cardTag, bs4_card_dep())
}
```

```
}
```

We finally run our app:

```
# run shiny app
ui <- fluidPage(
  title = "Hello Shiny!",
  fluidRow(
    column(
      width = 6,
      align = "center",
      br(),
      my_card("blablabla. PouetPouet Pouet.")
    )
  )
)

shinyApp(ui, server = function(input, output) { })
```

With this approach, you could develop a package of custom dependencies that people could use when they need to add custom elements in shiny.

### 6.3 Another example: Importing HTML dependencies from other packages

You may know shinydashboard, a package to design dashboards with shiny. In the following, we would like to integrate the box component in a classic Shiny App (without the dashboard layout). However, if you try to include the Shinydashboard box tag, you will notice that nothing is displayed since Shiny does not have shinydashboard dependencies. Fortunately htmltools contains a function, namely `findDependencies` that looks for all dependencies attached to a tag. How about extracting shinydashboard dependencies? Before going further, let's define the basic skeleton of a shinydashboard:

```
shinyApp(
  ui = dashboardPage(
    dashboardHeader(),
    dashboardSidebar(),
    dashboardBody(),
    title = "Dashboard example"
  ),
  server = function(input, output) { }
)
```

### 6.3. ANOTHER EXAMPLE: IMPORTING HTML DEPENDENCIES FROM OTHER PACKAGES 69

We don't need to understand shinydashboard details. However, if you are interested to dig in, help yourself. What is important here is the main wrapper function `dashboardPage`. (You should already be familiar with `fluidPage`, another wrapper function). We apply `findDependencies` on `dashboardPage`.

```
deps <- findDependencies(  
  dashboardPage(  
    header = dashboardHeader(),  
    sidebar = dashboardSidebar(),  
    body = dashboardBody()  
  )  
)  
deps
```

`deps` is a list containing 4 dependencies:

- Font Awesome handles icons
- Bootstrap is the main HTML/CSS/JS template. Importantly, please note the version 3.3.7, whereas the current is 4.3.1
- AdminLTE is the dependency containing HTML/CSS/JS related to the admin template. It is closely linked to Bootstrap 3.
- shinydashboard, the CSS and javascript necessary for shinydashboard to work properly. In practice, integrating custom HTML templates to shiny does not usually work out of the box for many reasons (Explain why!) and some modifications are necessary.

```
[[1]]  
List of 10  
 $ name      : chr "font-awesome"  
 $ version   : chr "5.3.1"  
 $ src       :List of 1  
 ..$ file: chr "www/shared/fontawesome"  
 $ meta      : NULL  
 $ script    : NULL  
 $ stylesheet: chr [1:2] "css/all.min.css" "css/v4-shims.min.css"  
 $ head      : NULL  
 $ attachment: NULL  
 $ package   : chr "shiny"  
 $ all_files : logi TRUE  
 - attr(*, "class")= chr "html_dependency"  
[[2]]  
List of 10  
 $ name      : chr "bootstrap"  
 $ version   : chr "3.3.7"  
 $ src       :List of 2
```

```

..$ href: chr "shared/bootstrap"
..$ file: chr "/Library/Frameworks/R.framework/Versions/3.5/Resources/library/shiny/www"
$ meta      :List of 1
..$ viewport: chr "width=device-width, initial-scale=1"
$ script     : chr [1:3] "js/bootstrap.min.js" "shim/html5shiv.min.js" "shim/respond.min.js"
$ stylesheet: chr "css/bootstrap.min.css"
$ head       : NULL
$ attachment: NULL
$ package    : NULL
$ all_files  : logi TRUE
- attr(*, "class")= chr "html_dependency"
[[3]]
List of 10
$ name       : chr "AdminLTE"
$ version    : chr "2.0.6"
$ src        :List of 1
..$ file: chr "/Library/Frameworks/R.framework/Versions/3.5/Resources/library/shinydashboard"
$ meta       : NULL
$ script     : chr "app.min.js"
$ stylesheet: chr [1:2] "AdminLTE.min.css" "_all-skins.min.css"
$ head       : NULL
$ attachment: NULL
$ package    : NULL
$ all_files  : logi TRUE
- attr(*, "class")= chr "html_dependency"
[[4]]
List of 10
$ name       : chr "shinydashboard"
$ version    : chr "0.7.1"
$ src        :List of 1
..$ file: chr "/Library/Frameworks/R.framework/Versions/3.5/Resources/library/shinydashboard"
$ meta       : NULL
$ script     : chr "shinydashboard.min.js"
$ stylesheet: chr "shinydashboard.css"
$ head       : NULL
$ attachment: NULL
$ package    : NULL
$ all_files  : logi TRUE
- attr(*, "class")= chr "html_dependency"

```

Below, we attach the dependencies to the `box` with `tagList`, as shown above. Notice that our custom `box` does not contain all parameters from `shinydashboard` but this is not what matters in this example.

```
my_box <- function(title, status) {  
  tagList(box(title = title, status = status), deps)  
}  
ui <- fluidPage(  
  titlePanel("Shiny with a box"),  
  my_box(title = "My box", status = "danger"),  
)  
server <- function(input, output) {}  
shinyApp(ui, server)
```

Now, you may imagine the possibilities are almost unlimited! Interestingly, this is the approach we use in `shinyWidgets` for the `useBs4Dash` function and other related tools.

## 6.4 Suppress dependencies

In rare cases, you may need to remove an existing dependency (conflict). The `suppressDependencies` function allows to perform that task. For instance, `shiny.semantic` built on top of `semantic ui` is not compatible with Bootstrap. Below, we remove the `AdminLTE` dependency from a `shinydashboard` page and nothing is displayed (as expected):

```
shinyApp(  
  ui = dashboardPage(  
    dashboardHeader(),  
    dashboardSidebar(),  
    dashboardBody(suppressDependencies("AdminLTE")),  
    title = "Dashboard example"  
  ),  
  server = function(input, output) { }  
)
```





## Chapter 7

# Other tools

### 7.1 CSS

- See `cascadess` to customize the style of tags

```
ui <- list(  
  cascadess(),  
  h4(  
    .style %>%  
      font(case = "upper") %>%  
      border(bottom = "red"),  
    "Etiam vel tortor sodales tellus ultricies commodo."  
  )  
)
```



# Practice



In this chapter, you will learn how to build your own html templates taken from the web and package them, so that they can be re-used at any time by anybody.



## Chapter 8

# Template selection

There exists tons of HTML templates over the web. However, only a few part will be suitable for shiny, mainly because of what follows:

- shiny is built on top of Bootstrap 3 (HTML, CSS and Javascript framework), meaning that going for another framework might not be straightforward. However, shinymaterial and shiny.semantic are good examples showing this can be possible.
- shiny relies on jQuery (currently v 3.4.1 for shiny). Consequently, all templates based upon React, Vue and other Javascript framework will not be natively supported. Again, there exist some examples for React with shiny and more generally, the reactR package developed by Kent Russell and Alan Dipert from RStudio.

See the github repository for more details about all dependencies related to the shiny package.

Notes: As shiny depends on Bootstrap 3.4.1, we recommend the user who would like to experiment Bootstrap 4 features to be particularly careful about potential incompatibilities. See a working example here with bs4Dash.

A good source of **open source** HTML templates is Colorlib and Creative Tim.

In the next chapter, we will focus on the tabler.io dashboard template (See Figure 8.1).

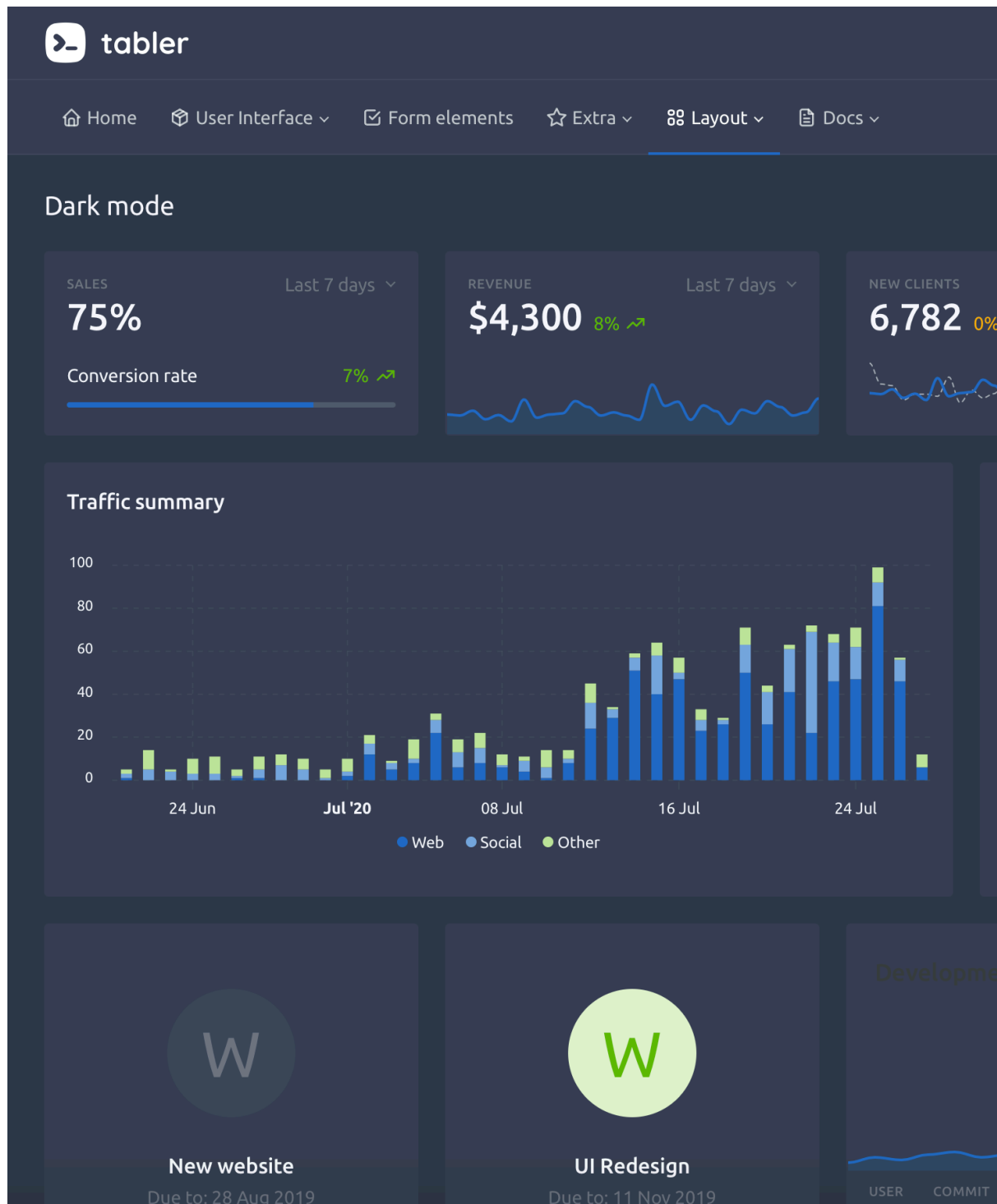


Figure 8.1: Tabler dashboard overview



## Chapter 9

# Define dependencies

This is time to start our practical session. As mentionned in the previous chapter, we will focus on the `tabler`, a tiny Bootstrap 4 dashboard template. Importantly, what follows is not the description of how to customize `tabler` but rather provide a R wrapper. Therefore we will not write SASS nore edit the core JavaScript, even though technically possible.

### 9.1 Discover the project

The first step of any template adaptation consists in exploring the underlying github repository (if open source) and look for mandatory elements, like CSS/JS dependencies. You would actually proceed similarly for an `HTMLWidget`.

As shown in Figure 9.1, the most important folders are:

- `dist`: contains CSS and JS files as well as other libraries like Bootstrap and jQuery. In production we seek for minified files since they take less space. It is also a good moment to look at the version of each dependency that might conflict with Shiny
- `demo` is the website folder used for demonstration purpose. This is our source to explore the template capabilities in depth

The `scss` and `build` folder are also crucial to the package but as stated previously, customizing `tabler` is out of the scope of this book. It is already a significant task to adapt a template from a language to another.












 <b>.dependabot</b>	dependabot update
 <b>.github</b>	Merge pull request
 <b>build</b>	change-version scr
 <b>demo</b>	1.0.0-alpha.7
 <b>dist</b>	1.0.0-alpha.7
 <b>img</b>	buttons, payments,
 <b>js</b>	navbar overlap, wel
 <b>pages</b>	bootstrap upgrade,
 <b>scss</b>	bootstrap upgrade,
 <b>static</b>	navbar overlap, wel
 <b>svg/brand</b>	svg icons incremen

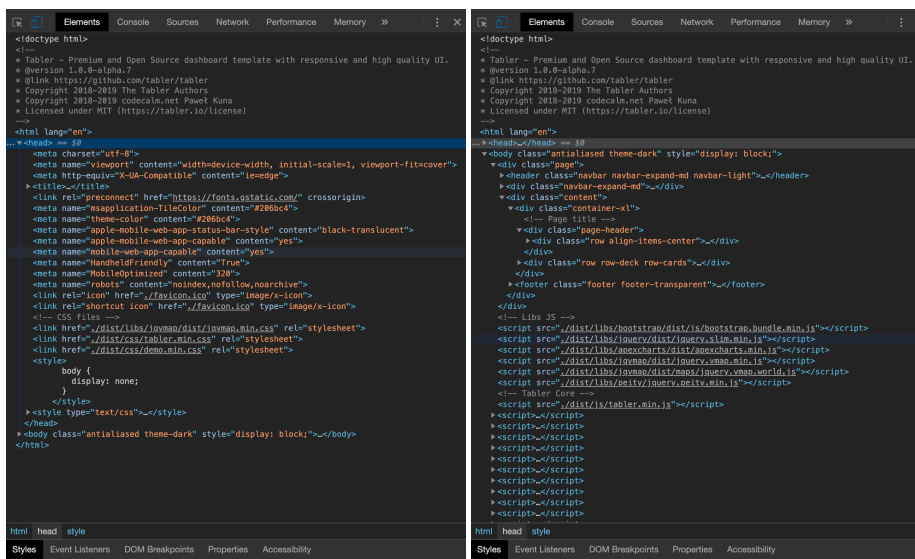
Figure 9.1: Github project exploration

## 9.2 Identify mandatory dependencies

Now, among all JS/CSS resources, we need to identify the one necessary to the template. Obviously, the Bootstrap 4, jQuery, tabler.min.css and tabler.min.js are key elements, contrary to flag icons which are optional (and take a lot of space). The package size is also to consider if you plan to release the template on CRAN and respect the 5mB maximum limit. By experience, I can tell you this is quite hard to handle.

To inspect dependencies, we proceed as follows

- Download or clone the github repository
- Go to the demo folder and open the layout-dark.html file
- Open the HTML inspector



As depicted on Figure ?? left-hand side, we need to include the tabler.min.css from the header. If you are not convinced, try to remove it from the DOM and see what happens. jqvmap is actually related to an external visualization plugin used in the demo. Finally the demo.min.css file is for the demo purpose. This will not prevent the template to work if we skip it. So far so good, we only need 1 file thus!

JavaScript dependencies are shown on the right-hand side and located at the end of the body tag. We won't need all chart-related dependencies like apexcharts, jquery.vmap and vmap world and can safely ignore them. We will keep the Bootstrap 4 bundle.js, jQuery core and tabler.min.js (the order is crucial).

### 9.3 Bundle dependencies

With the help of the `htmltoolsDependency` function, we are going to create our main `tabler` HTML dependency containing all assets to allow our template to render properly. In this example, I am going to cheat a bit: instead of handling local files, I will use a CDN (content delivery network) that hosts all necessary `tabler` assets. The main reason is that it will allow us to test the template directly from the `bookdown` project. But in theory, this template would need to live inside an R package, in a github repository.

```
tablers_deps <- htmlDependency(
  name = "tabler",
  version = "1.0.7", # we take that of tabler,
  src = c(href = "https://cdn.jsdelivr.net/npm/tabler@1.0.0-alpha.7/dist/"),
  script = "js/tabler.min.js",
  stylesheet = "css/tabler.min.css"
)
```

I advise the reader to create 1 HTML dependency per element. The `Bootstrap` version is `v4.3.1` (`Shiny` relies on `3.4.1`) and `jQuery` is `3.5.0` (`Shiny` relies on `3.4.1`). We can also use a CDN:

```
bs4_deps <- htmlDependency(
  name = "Bootstrap",
  version = "4.3.1",
  src = c(href = "https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/"),
  script = "bootstrap.bundle.min.js"
)

jQuery_deps <- htmlDependency(
  name = "jquery",
  version = "3.5.0",
  src = c(href = "https://code.jquery.com/"),
  script = "jquery-3.5.0.slim.min.js"
)
```

We finally create our dependency manager (TO DO: add more details):

```
# add all dependencies to a tag. Don't forget to set append to TRUE to preserve any ex
addDeps <- function(tag) {
  # below, the order is of critical importance!
  deps <- list(bs4_deps, tablers_deps)
  attachDependencies(tag, deps, append = TRUE)
}
```

Let's see how to use `addDeps`. We consider a `<div>` placeholder and check for its dependencies with `findDependencies` (should be `NULL`). Then, we wrap it with `addDeps`.

```
tag <- div()
findDependencies(tag)
```

```
## NULL
```

```
tag <- addDeps(div())
findDependencies(tag)
```

```
## [[1]]
## List of 10
## $ name      : chr "Bootstrap"
## $ version   : chr "4.3.1"
## $ src       :List of 1
## ..$ href: chr "https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/"
## $ meta      : NULL
## $ script    : chr "bootstrap.bundle.min.js"
## $ stylesheet: NULL
## $ head      : NULL
## $ attachment: NULL
## $ package   : NULL
## $ all_files : logi TRUE
## - attr(*, "class")= chr "html_dependency"
##
## [[2]]
## List of 10
## $ name      : chr "tabler"
## $ version   : chr "1.0.7"
## $ src       :List of 1
## ..$ href: chr "https://cdn.jsdelivr.net/npm/tabler@1.0.0-alpha.7/dist/"
## $ meta      : NULL
## $ script    : chr "js/tabler.min.js"
## $ stylesheet: chr "css/tabler.min.css"
## $ head      : NULL
## $ attachment: NULL
## $ package   : NULL
## $ all_files : logi TRUE
## - attr(*, "class")= chr "html_dependency"
```

As shown above, our dependencies are applied to the `div`, in the correct order. This order is set by the list `list(bs4_deps, jQuery_deps, tablers_deps)`.

This flexibility allows to avoid potential conflicts. If we try to run this simple tag in a shiny app, we notice that all dependencies are added to the `<head>` tag, whereas the original template loads JavaScript dependencies in the `<body>`. Currently, `htmltools` does not allow to distribute dependencies in different places. Here there is no impact but for other templates like `Framework7` (which is powering `shinyMobile`), JavaScript must be place in the body. In practice, this is quite honestly hard to guess and only manual testing will help you.

```
ui <- fluidPage(tag)
server <- function(input, output, session) {}
shinyApp(ui, server)
```

Even though the `addDeps` function may be applied to any tag, we will use it with the core HTML template, that remain to be designed!

Would you like to see if our dependency system works? Let's meet in the next chapter to design the main dashboard layout.

## Chapter 10

# Template skeleton

The list of all available layouts is quite impressive (horizontal, vertical, compressed, right to left, dark, ...). In the next steps, we will focus on the dark-compressed template. We leave the reader to try other templates as an exercise.

### 10.1 Identify template elements

We are quite lucky since there is nothing fancy about the tabler layout. As usual, let's inspect the layout-condensed-dark.html (in the tabler /demo folder) in Figure 10.1

There are 2 main components: - the header containing the brand logo, the navigation and dropdown - the content containing the dashboard body as well as the footer

Something important: the dashboard body does not mean `<body>` tag!

This is all!

### 10.2 Design the page layout

#### 10.2.1 The page wrapper

Do you remember the structure of a basic html page seen in Chapter 2? Well, if not, here is a reminder.

```
<!DOCTYPE HTML>  
<html>
```

```

<!doctype html>
<!--
* Tabler - Premium and Open Source dashboard template with responsive
* @version 1.0.0-alpha.7
* @link https://github.com/tabler/tabler
* Copyright 2018-2019 The Tabler Authors
* Copyright 2018-2019 codecalm.net Paweł Kuna
* Licensed under MIT (https://tabler.io/license)
-->
<html lang="en">
... ▶ <head>...</head> == $0
▼ <body class="antialiased" style="display: block;">
  ▼ <div class="page">
    ▼ <header class="navbar navbar-expand-md navbar-dark">
      ▼ <div class="container-xl">
        ▶ <button class="navbar-toggler" type="button" data-toggle="col
"#navbar-menu">...</button>
        ▶ <a href="." class="navbar-brand navbar-brand-autodark d-none-
pr-md-3">...</a>
        ▶ <div class="navbar-nav flex-row order-md-last">...</div>
        ▼ <div class="collapse navbar-collapse" id="navbar-menu">
          ▼ <div class="d-flex flex-column flex-md-row flex-fill align-
items-md-center">
            ▶ <ul class="navbar-nav">...</ul>
            ▶ <div class="ml-md-auto pl-md-4 py-2 py-md-0 mr-md-4 order-
flex-grow-1 flex-md-grow-0">...</div>
          </div>
        </div>
      </div>
    </header>
    ▼ <div class="content">
      ▼ <div class="container-xl">
        <!-- Page title -->
        ▶ <div class="page-header">...</div>
        ▶ <div class="row row-deck row-cards">...</div>
      </div>
      ▶ <footer class="footer footer-transparent">...</footer>
    </div>
  </div>

```

Figure 10.1: Tabler condensed layout



```

<head>
  <!-- head content here -->
</head>
<body>
  <p>Hello World</p>
</body>
</html>

```

We actually don't need to take care of the `<html>` tag. Below we construct a list of tags with `tagList`, including the head and the body. In the head we have `meta` tag that briefly describe the encoding, how to display the app on different devices (For instance `apple-mobile-web-app-status-bar-style` is for mobile support), set the favicon (website icon, the icon you see on the right side of the searchbar. Try twitter for instance). The page title may change, so is the favicon, so we include them as parameters of the function. If you remember, there also should be CSS in the head but nothing here! Actually, the insertion of dependencies will be achieved by our `addDeps` function defined in Chapter 9. This is what we do to the `<body>` tag that is wrapped by this function. Let's talk about the `dark` parameter. In short, the only difference between the dark and the light theme is the class applied to the `<body>` tag (respectively "antialiased theme-dark" and "antialiased"). The ... parameter contain other template elements like the header and the dashboard body, that remain to be created.

```

tabler_page <- function(..., dark = TRUE, title = NULL, favicon = NULL){

  tagList(
    # Head
    tags$head(
      tags$meta(charset = "utf-8"),
      tags$meta(
        name = "viewport",
        content = "
          width=device-width,
          initial-scale=1,
          viewport-fit=cover"
      ),
      tags$meta(`http-equiv` = "X-UA-Compatible", content = "ie=edge"),
      tags$title(title),
      tags$link(
        rel = "preconnect",
        href = "https://fonts.gstatic.com/",
        crossorigin = NA
      ),
      tags$meta(name = "msapplication-TileColor", content = "#206bc4"),

```

```

tags$meta(name = "theme-color", content = "#206bc4"),
tags$meta(name = "apple-mobile-web-app-status-bar-style", content = "black-transp
tags$meta(name = "apple-mobile-web-app-capable", content = "yes"),
tags$meta(name = "mobile-web-app-capable", content = "yes"),
tags$meta(name = "HandheldFriendly", content = "True"),
tags$meta(name = "MobileOptimized", content = "320"),
tags$meta(name = "robots", content = "noindex,nofollow,noarchive"),
tags$link(rel = "icon", href = favicon, type = "image/x-icon"),
tags$link(rel = "shortcut icon", href = favicon, type="image/x-icon")
),
# Body
addDeps(
  tags$body(
    tags$div(
      class = paste0("antialiased ", if(dark) "theme-dark"),
      style = "display: block;",
      tags$div(class = "page", ...)
    )
  )
)
}

```

Below we quickly test if a `tabler` element renders well to see whether our dependency system is adequately setup. To that end, we include a random `tabler` element taken from the demo html page and include it as raw html, using `HTML`. We also ensure that basic Shiny input/output system works as expected with a `sliderInput` linked to a plot output.

```

ui <- tabler_page(
  "test",
  sliderInput("obs", "Number of observations:",
    min = 0, max = 1000, value = 500
  ),
  plotOutput("distPlot"),
  br(),
  HTML(
    '<div class="col-sm-6 col-lg-3">
    <div class="card">
      <div class="card-body">
        <div class="d-flex align-items-center">
          <div class="subheader">Sales</div>
          <div class="ml-auto lh-1">
            <div class="dropdown">
              <a class="dropdown-toggle text-muted" href="#" data-toggle="dropdown

```

```

        Last 7 days
      </a>
      <div class="dropdown-menu dropdown-menu-right">
        <a class="dropdown-item active" href="#">Last 7 days</a>
        <a class="dropdown-item" href="#">Last 30 days</a>
        <a class="dropdown-item" href="#">Last 3 months</a>
      </div>
    </div>
  </div>
</div>
<div class="h1 mb-3">75%</div>
<div class="d-flex mb-2">
  <div>Conversion rate</div>
  <div class="ml-auto">
    <span class="text-green d-inline-flex align-items-center lh-1">
      7%
      <svg xmlns="http://www.w3.org/2000/svg" class="icon ml-1" width="24" height="24">
        <path stroke="none" d="M0 0h24v24H0z"></path>
        <polyline points="3 17 9 11 13 15 21 7"></polyline>
        <polyline points="14 7 21 7 21 14"></polyline>
      </svg>
    </span>
  </div>
</div>
<div class="progress progress-sm">
  <div class="progress-bar bg-blue" style="width: 75%" role="progressbar" aria-valuenow="75">
    <span class="sr-only">75% Complete</span>
  </div>
</div>
</div>
</div>
</div>
),
title = "Tabler test"
)
server <- function(input, output, session) {
  output$distPlot <- renderPlot({
    hist(rnorm(input$obs))
  })
}
shinyApp(ui, server)

```

Ok... The layout is ugly, margins are not correct, the plot background does not match with the overall theme, ... but our info card and the shiny element work

like a charm, which is a good start.

### 10.2.2 The body content

In this part, we translate the dashboard body HTML code to R. We create a function called `tabler_body`. The `...` parameter holds all the dashboard body elements and the footer is dedicated for the future `tabler_footer` function.

```
tabler_body <- function(..., footer = NULL) {
  div(
    class = "content",
    div(class = "container-xl", ...),
    tags$footer(class = "footer footer-transparent", footer)
  )
}
```

Let's test it with the previous example.

```
ui <- tabler_page(tabler_body(p("Hello World")))
server <- function(input, output, session) {}
shinyApp(ui, server)
```

Way better!

### 10.2.3 The footer

The footer is composed of a left and right containers. We decide to create parameters `left` and `right` in which the user will be able to pass any elements.

```
tabler_footer <- function(left = NULL, right = NULL) {
  div(
    class = "container",
    div(
      class = "row text-center align-items-center flex-row-reverse",
      div(class = "col-lg-auto ml-lg-auto", right),
      div(class = "col-12 col-lg-auto mt-3 mt-lg-0", left)
    )
  )
}
```

As above, let's check our brand new element.

```

ui <- tabler_page(
  tabler_body(
    p("Hello World"),
    footer = tabler_footer(
      left = "Rstats, 2020",
      right = a(href = "https://www.google.com")
    )
  )
)
server <- function(input, output, session) {}
shinyApp(ui, server)

```

### 10.2.4 The navbar (or header)

This function is called `tabler_header`. In the tabler template, the header has the classes “`navbar navbar-expand-md navbar-light`”. We don’t need the `navbar-light` class since we are interested in the dark theme. As shown in Figure 10.2, the navbar is composed of 4 elements:

- the navbar toggler is only visible when we reduce the screen width (like on mobile devices)
- the brand image
- the navigation
- the dropdown menu (this is not mandatory)

You may have a look at the Bootstrap 4 documentation for the navbar configuration and layout.

Each of these element will be considered as an input parameter to the `tabler_navbar` function, except the first element which is a default element and should not be removed. Moreover, we will only show the brand element when it is provided. The `...` parameter is a slot for extra elements (between the menu and dropdowns).

```

tabler_navbar <- function(..., brand_url = NULL, brand_image = NULL, nav_menu, nav_right = NULL)
  navbar_cl <- "navbar navbar-expand-md"
  tags$header(
    class = navbar_cl,
    tags$div(
      class = "container-xl",
      # toggler for small devices (must not be removed)
      tags$button(
        class = "navbar-toggler",
        type = "button",

```

```

▼ <header class="navbar navbar-expand-md navbar-dark">
  ▼ <div class="container-xl"> = $0
    ▶ <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbar-menu">...</button>
    ▶ <a href="." class="navbar-brand navbar-brand-autodark d-none-navbar-horizontal pr-0 pr-md-3">...</a>
    ▶ <div class="navbar-nav flex-row order-md-last">...</div>
    ▶ <div class="collapse navbar-collapse" id="navbar-menu">...</div>
  </div>
</header>

```

Figure 10.2: Tabler header structure

```

    `data-toggle` = "collapse",
    `data-target` = "#navbar-menu",
    span(class = "navbar-toggler-icon")
  ),

  # brand stuff
  if (!is.null(brand_url) || !is.null(brand_image)) {
    a(
      href = if (!is.null(brand_url)) {
        brand_url
      } else {
        "#"
      },
      class = "navbar-brand navbar-brand-autodark d-none-navbar-horizontal pr-0 pr-md-3",
      if(!is.null(brand_image)) {
        img(
          src = brand_image,
          alt = "brand Image",
          class = "navbar-brand-image"
        )
      }
    )
  },

  # slot for dropdown element
  if (!is.null(nav_right)) {
    div(class = "navbar-nav flex-row order-md-last", nav_right)
  },

  #

```

```

div(
  class = "collapse navbar-collapse",
  id = "navbar-menu",
  div(
    class = "d-flex flex-column flex-md-row flex-fill align-items-stretch align-items-md-center",
    nav_menu
  ),
  if (length(list(...)) > 0) {
    div(
      class = "ml-md-auto pl-md-4 py-2 py-md-0 mr-md-4 order-first order-md-last flex-grow-1",
      ...
    )
  }
)
)
}

```

Let's create the navbar menu. The ... parameter is a slot for the menu items. Compared to the original tabler dashboard template where there is only the class navbar-nav, we have to add, at least, the nav class to make sure items are correctly activated/inactivated. The nav-pills class is to select pills instead of basic tabs (see here).

```

tabler_navbar_menu <- function(...) {
  tags$ul(class = "nav nav-pills navbar-nav", ...)
}

```

Each navbar menu item could be either a simple button or contain multiple menu sub-items. For now, we only focus on simple items.

#### 10.2.4.1 Navbar navigation

This part is extremely important since it will drive the navigation of the template. What do we want? We would like to associate each item to a separate page in the body content, so that each time we change item, we go on another page. In brief, it is very similar to the Shiny `tabsetPanel` function.

In HTML, menu items are `<a>` tags (links) with a given `href` attribute pointing to a specific page located in the server files. The point with a Shiny app is that we can't decide how to split our content into several pages. We only have app.R generating a simple HTML page. The strategy here is to create a tabbed navigation, to mimic multiple pages.

Let's see how tabset navigation works. In the menu list, all items must have a `data-toggle` attribute set to tab, an `href` attribute holding a unique id. This

unique id is mandatory since it will point the menu item to the corresponding body content. On the body side, tab panels are contained in a tabset panel (simple div container), have a `role` attribute set to `tabpanel` and an `id` corresponding the `tabName` passed in the menu item. Below, we propose a possible implementation of a menu item, as well as the corresponding body tab panel.

```
tabler_navbar_menu_item <- function(text, tabName, icon = NULL, selected = FALSE) {

  item_cl <- paste0("nav-link", if(selected) " active")

  tags$li(
    class = "nav-item",
    a(
      class = item_cl,
      href = paste0("#", tabName),
      `data-toggle` = "pill", # see https://getbootstrap.com/docs/4.0/components/navs/
      `data-value` = tabName,
      role = "tab",
      span(class = "nav-link-icon d-md-none d-lg-inline-block", icon),
      span(class = "nav-link-title", text)
    )
  )
}
```

We also decided to add a fade transition effect between tabs, as per Bootstrap 4 documentation.

```
tabler_tab_items <- function(...) {
  div(class = "tab-content", ...)
}

tabler_tab_item <- function(tabName = NULL, ...) {
  div(
    role = "tabpanel",
    class = "tab-pane fade container-fluid",
    id = tabName,
    ...
  )
}
```

What about testing this in a shiny app?

```
ui <- tabler_page(
  tabler_navbar(
    brand_url = "https://preview-dev.tabler.io",
```



```

brand_image = "https://preview-dev.tabler.io/static/logo.svg",
nav_menu = tabler_navbar_menu(
  tabler_navbar_menu_item(
    text = "Tab 1",
    icon = NULL,
    tabName = "tab1",
    selected = TRUE
  ),
  tabler_navbar_menu_item(
    text = "Tab 2",
    icon = NULL,
    tabName = "tab2"
  )
),
tabler_body(
  tabler_tab_items(
    tabler_tab_item(
      tabName = "tab1",
      p("Hello World")
    ),
    tabler_tab_item(
      tabName = "tab2",
      p("Second Tab")
    )
  ),
  footer = tabler_footer(
    left = "Rstats, 2020",
    right = a(href = "https://www.google.com")
  )
)
)
server <- function(input, output, session) {}
shinyApp(ui, server)

```

#### 10.2.4.2 Fine tune tabs behavior

Quite good isn't it? However, you will notice that even if the first tab was selected by default, its content is not shown! To fix this, we will apply our jQuery skills. According to the Bootstrap documentation, we must trigger the `show` event on the active tab at start, as well as add the classes `show` and `active` to the associated tab panel in the dashboard body. We therefore target the nav item that has the active class and if no item is found, we select the first item by default and activate its tab.

```
$(function() {
  // this makes sure to trigger the show event on the active tab at start
  const activeTab = $('#navbar-menu .nav-link.active');
  // if multiple items are found
  if (activeTab.length > 0) {
    const tabId = $(activeTab).attr('data-value');
    $(activeTab).tab('show');
    $('#${tabId}`).addClass('show active');
  } else {
    $('#navbar-menu .nav-link')
      .first()
      .tab('show');
  }
});
```

The script is included in a tag but best practice it to put it in a separate js file (I do it this way because it is more convenient for the demonstration).

```
ui <- tabler_page(
  tags$head(
    tags$script(
      HTML(
        "$(function() {
          // this makes sure to trigger the show event on the active tab at start
          const activeTab = $('#navbar-menu .nav-link.active');
          if (activeTab.length > 0) {
            const tabId = $(activeTab).attr('data-value');
            $(activeTab).tab('show');
            $('#${tabId}`).addClass('show active');
          } else {
            $('#navbar-menu .nav-link')
              .first()
              .tab('show');
          }
        });
      "
    )
  ),
  tabler_navbar(
    brand_url = "https://preview-dev.tabler.io",
    brand_image = "https://preview-dev.tabler.io/static/logo.svg",
    nav_menu = tabler_navbar_menu(
      tabler_navbar_menu_item(
        text = "Tab 1",
```

```

        icon = NULL,
        tabName = "tab1",
        selected = TRUE
      ),
      tabler_navbar_menu_item(
        text = "Tab 2",
        icon = NULL,
        tabName = "tab2"
      )
    ),
    tabler_body(
      tabler_tab_items(
        tabler_tab_item(
          tabName = "tab1",
          p("Hello World")
        ),
        tabler_tab_item(
          tabName = "tab2",
          p("Second Tab")
        )
      ),
      footer = tabler_footer(
        left = "Rstats, 2020",
        right = a(href = "https://www.google.com")
      )
    )
  )
server <- function(input, output, session) {}
shinyApp(ui, server)

```

The result is shown in Figure 10.3. I'd also suggest to include at least 1 input/output per tab, to test whether everything works properly.

Looks like we are done for the main template elements. Actually, wouldn't it be better to include, at least, card containers?

### 10.2.5 Card containers

Card are a central piece of template as they may contain visualizations, metrics and much more. Tabler has a large range of card containers.

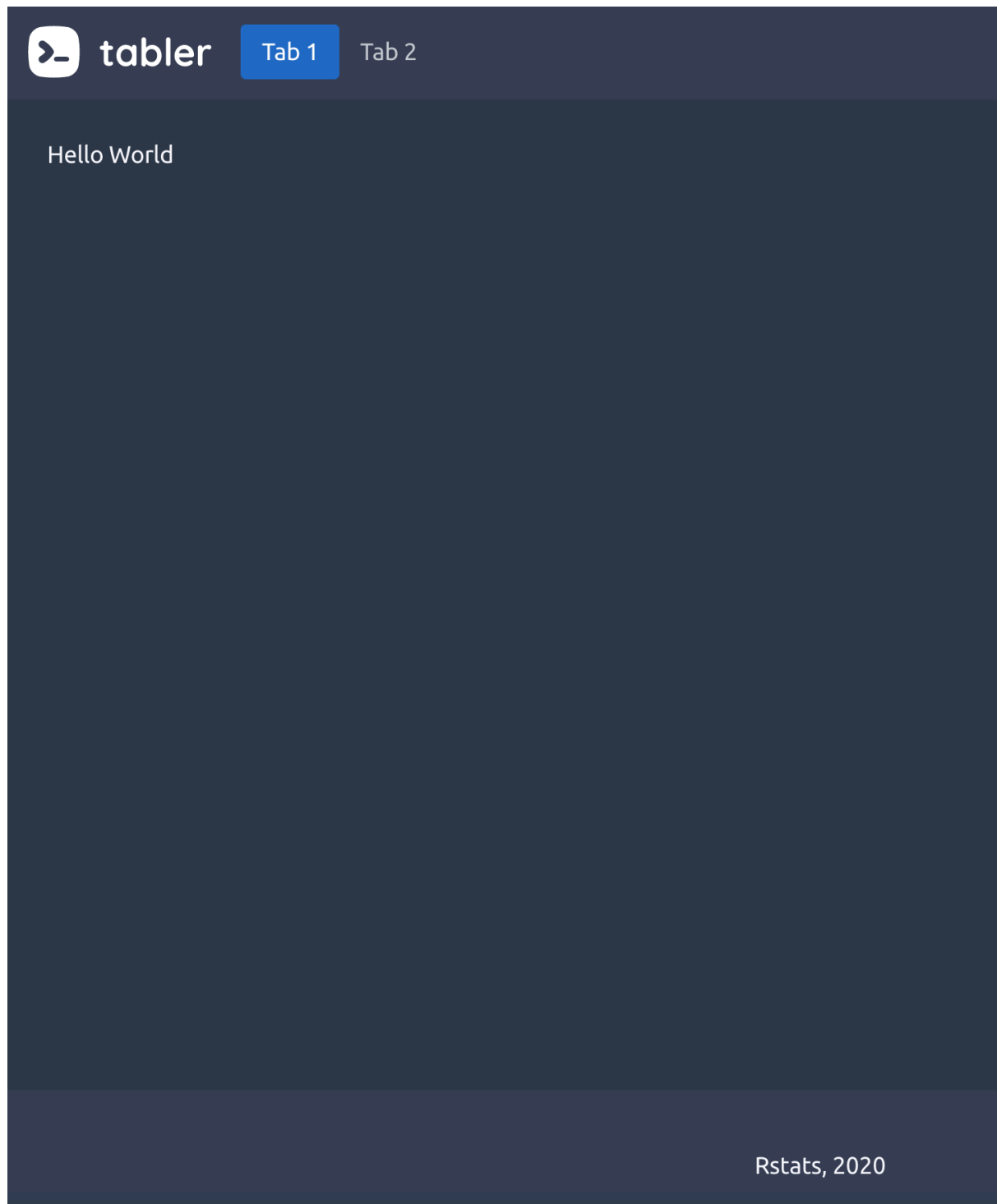


Figure 10.3: Tabler template with navbar

### 10.2.5.1 Classic card

What I call a classic card is like the `box` container of shinydashboard. The card structure has key elements:

- a width to control the space taken by the card in the Bootstrap grid
- a title, in general in the header (tabler does always not follow this rule and header is optional)
- a body where is the main content
- style elements like color statuses
- a footer (optional, tabler does not include this)

A comprehensive list of all tabler card features may be found here. To be faster, I will copy the following HTML code in the html2R shiny app to convert it to Shiny tags

```
<div class="col-md-6">
  <div class="card">
    <div class="card-status-top bg-danger"></div>
    <div class="card-body">
      <h3 class="card-title">Title</h3>
      <p>Some Text.</p>
    </div>
  </div>
</div>
```

Below is the result. The next step consist in replacing all content by parameters to the `tabler_card` function, whenever necessary. For instance, the first `<div>` sets the width of the card. The Bootstrap grid ranges from 0 to 12, so why not creating a width parameter to control the card size. We proceed similarly for the title, status, body content. A last comment on parameters default values. It seems reasonable to allow title to be NULL (if so, the title won't be shown), same thing for the status. Regarding the card default width, 6 also makes sense.

```
tabler_card <- function(..., title = NULL, status = NULL, width = 6, stacked = FALSE, padding = NULL) {

  card_cl <- paste0(
    "card",
    if (stacked) " card-stacked",
    if (!is.null(padding)) paste0(" card-", padding)
  )

  div(
    class = paste0("col-md-", width),
    div(
```

```

    class = card_cl,
    if (!is.null(status)) {
      div(class = paste0("card-status-top bg-", status))
    },
    div(
      class = "card-body",
      # we could have a smaller title like h4 or h5...
      if (!is.null(title)) {
        h3(class = "card-title", title)
      },
      ...
    )
  )
)
}

# test the card
my_card <- tabler_card(
  p("Hello"),
  title = "My card",
  status = "danger"
)

```

In the meantime, I'd be also nice to be able to display cards in the same row. Let's create the `tabler_row`:

```

tabler_row <- function(...) {
  div(class = "row row-deck", ...)
}

```

```

ui <- tabler_page(
  tabler_body(
    tabler_row(
      my_card,
      tabler_card(
        p("Hello"),
        title = "My card",
        status = "success"
      )
    )
  )
)
server <- function(input, output, session) {}
shinyApp(ui, server)

```

### 10.2.6 Ribbons: card components

Let's finish this part by including a card component, namely the ribbon.

```
tabler_ribbon <- function(..., position = NULL, color = NULL, bookmark = FALSE) {

  ribbon_cl <- paste0(
    "ribbon",
    if (!is.null(position)) sprintf(" bg-%s", position),
    if (!is.null(color)) sprintf(" bg-%s", color),
    if (bookmark) " ribbon-bookmark"
  )
  div(class = ribbon_cl, ...)
}
```

Integrating the freshly created ribbon component requires to modify the card structure since the ribbon is added after the body tag, and not parameter is associated with this slot. We could also modify the `tabler_card` function but `htmltools` contains tools to help us. Since the ribbon should be put after the card body (but in the card container), we may think about the `tagAppendChild` function, introduced in Chapter 5:

```
# add the ribbon to a card
my_card <- tabler_card(title = "Ribbon")
my_card$children[[1]] <- my_card$children[[1]] %>%
  tagAppendChild(
    tabler_ribbon(
      icon("info-circle", class = "fa-lg"),
      bookmark = TRUE
    )
  )
```

As shown above, the ribbon has been successfully included in the card tag. Now, we check how it looks in a shiny app.

```
ui <- tabler_page(
  tabler_body(
    my_card
  )
)
server <- function(input, output, session) {}
shinyApp(ui, server)
```

### 10.2.7 Icons

Not mentionned before but we can use fontawesome icons provided with Shiny, as well as other libraries. Moreover, tabler has a svg library located [here](#).



## Chapter 11

# Develop custom input widgets

11.1 How does Shiny handle inputs?

11.2 How to add new input to Shiny?



## Chapter 12

# Testing templates elements